DESIGN OF AN AUTONOMOUS ANTI-DDOS NETWORK (A2D2)

by

ANGELA CEARNS, B.A.

University of Western Ontario, London, Ontario, Canada, 1995

A Thesis

Submitted to the Faculty of Graduate School of the

University of Colorado at Colorado Springs

in Partial Fulfillment of the Requirements

for the Degree of

Master of Engineering

Department of Computer Science

2002

This thesis for the Master of Engineering degree by

Angela Cearns

has been approved for the

Department of Computer Science

by

Advisor: Dr. C. Edward Chow

Dr. Jugal K. Kalita

Dr. Charles M. Shub

Date

Design of an Autonomous Anti-DDoS Network (A2D2)

by

Angela Cearns

(Master of Engineering, Software Engineering)

Thesis directed by Associate Professor C. Edward Chow

Department of Computer Science

Abstract

Recent threats of Distributed Denial of Service attacks (DDoS) are mainly directed at home and small to medium sized networks that lack the incentive, expertise, and financial means to defend themselves. Using the Evolutionary Software Life-Cycle model, this thesis designs an Autonomous Anti-DDoS Network (A2D2) that integrates and improves on existing DDoS mitigation technologies. A2D2 provides an affordable and manageable solution to small and medium networks, and enables small office and home office (SOHO) networks to take control of their own defense within their own network boundary. Test-bed results show that A2D2 is highly effective in ensuring Quality of Service (QoS) during bandwidth consumption DDoS attacks. The A2D2 test-bed has demonstrated significant intrusion tolerance against attacks of various types, including UDP, ICMP and TCP based DDoS attacks.

This thesis is dedicated to my best friend and soul mate,

Kevin

Acknowledgements

Many people have shared their time and expertise to help me accomplish my goal. First, I would like to sincerely thank my advisor, Dr. C. Edward Chow, for his constant support and guidance. His worldwide around-the-clock instant responses to my countless inquiries have been invaluable and motivational. Many thanks to Dr. Jugal K. Kalita, who first introduced me to the concept of DDoS and whose lectures have inspired me to choose the current research focus. Thanks to Dr. Charles M. Shub for his reassurance at times of frustration. I am grateful to Dave Lohmann for troubleshooting my network and providing priceless support during my testing. Special thanks to Karen Hill, Dean Salsberg, Mary Lou and Michael Sharp, who help edit this thesis. Also deserve much credit are my many friends, the Snort community, Rhea, Nicole, Jim, and the staff in the Computer Science department.

I wish to pay special tributes to two much-cherished friends, Ganesh Kumar Godavari and Kaushal S.C. Joshy, who sacrificed the lovely summer months to assist and accompany me in the enclosed computer lab. I remain in awe with Ganesh's technical superiority and in deep appreciation for his willingness to share his knowledge. Without his help, I would probably still be in the midst of learning network programming. Thanks to Kaushal for his help on Perl programming. His patience and "Chai" have kept me reasonably pleasant in trying times.

Finally, I need to acknowledge two very important persons in my life, my husband Kevin and my "mom" and dearest friend - Janice L. Brittain. Without their love, support, friendship, and faith, I will not be here today. Thank You.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Since early 2000 when a number of high profile sites such as eBay and Yahoo.com were halted by Distributed Denial of Service (DDoS) attacks [Dit00], the initial furor has subsided but the continual threat has ascended. The prevalence of DDoS attacks was verified by a recent study conducted by the University of California, San Diego (UCSD), that detected approximately 12,805 Denial of Service (DoS) attacks against more than 5,000 targets during a three-week period in mid-2001 [MVS01]. Even CERT, the authority that warns Internet users on security threats, fell victim to DDoS in May 2001 [ITW01]. The Computer Security Institute (CSI) and the Federal Bureau of Investigation (FBI) reported in April 2002 that attacks had continued to "climb for third year in a row", with 40 percent of the 503 surveyed corporations, government agencies and universities detecting DoS attacks in 2002 [CSI02].

The increase of DoS attacks can partly be attributed to the development of more sophisticated and "user-friendly" tools such as Trinoo [Dit99], Trible Flood

Network (TFN)[Dit99], and Stacheldraht[Dit99]. Such tools have amplified the impact of DoS attacks exponentially through distributed channels and created a new variation of attacks called DDoS. The following sections describe the various types of DoS attacks, how these attacks evolve into DDoS attacks and what popular products are in the market that defend against DoS and DDoS attacks are.

1.1 Denial of Service Attack (DoS)

One widely accepted definition of computer security is the attainment of confidentiality, integrity, and availability in a computer system [RS91]. Confidentiality refers to the protection of information from unauthorized access. Integrity prevents data from being altered accidentally or by malicious attempts. Availability ensures the provision of quality resources and service to users as needed. DoS attacks target the availability aspect of computer security and deny services or resources to legitimate users. According to CERT, DoS attacks can be categorized into three types:

- Consumption of scarce, limited, or non-renewable resources
- Destruction or alteration of configuration information
- Physical destruction or alteration of network components [CERT01]

1.1.1 DoS – Consumption of Limited Resources

The operations of computers and networks rely on the availability of various resources such as network bandwidth, data structures, disk space, and power supply. A consumption DoS attack may be executed against any resource. For example, a TCP half-open attack consumes the kernel data structures involved in establishing a TCP network connection [Comer00]. When a computer receives a TCP SYN connection request, it responds with a SYN-ACK acknowledgement packet while keeping track of the SYN request in the kernel data structure. Such a TCP connection is considered half-opened and will remain open until the computer receives an ACK response from the initiating host completing the connection. A TCP half-open attacker can initiate enough half-open connections so that the victim machine is left with no data structure to service additional connection requests from legitimate clients.

In a bandwidth-consumption attack, an intruder directs a large number of packets towards your network, thereby consuming all the available bandwidth and stopping legitimate packets from accessing the congested network link. An attacker can also consume other resources such as disk space by generating excessive numbers of mail messages.

1.1.2 DoS – Destruction or Alteration of Configuration Information

In such attacks, an intruder may modify or destroy the system configuration information of the computer, such as the registry on a Windows NT machine or the routing information of routers. Such alteration may render certain functions unavailable, cause the systems to crash, or disable network connection by routing all packets to the attacker's server instead of the legitimate server.

1.1.3 DoS – Physical Destruction or Alteration of Computer Components

Perhaps the first DoS attack in history is the physical destruction of a computer component where an intruder disconnects power supply, slivers wires, and disables cooling stations so that the systems cannot service the clients.

1.2 Distributed Denial of Service Attack (DDoS)

Distributed Denial of Service (DDoS) attacks are any DoS attacks where tools are employed to rapidly "recruit" and coordinate attacks using a mass number of conspirators from widely diverse systems around the globe. A detailed account of the evolution of DDoS tools is provided by Dave Dittrich at the University of Washington [Dit00]. Dave Dittrich also conducted comprehensive analyses of such DDoS tools as Trinoo, TFN, Stacheldraht, and mstream [Dit99]. According to Dittrich, a DDoS attack is conducted in two phases: initial intrusion (Phase 1) and the DoS attack (Phase 2) [Dit00].

During the initial intrusion phase, the mastermind intruder identifies compromised hosts using vulnerability-scanning tools such as nmap [Nmap02] and whisker [Whi00]. To further distance his or her involvement, the chief intruder will organize the compromised systems into various categories with different functions assigned to them. A compromised system may be designated as an attack commander or a "client" who contains the blue-print of the attack plan. A few compromised systems will be defined as the "handlers" or the middlemen who coordinate the attack agents, while a vast number of systems will take on the role as the "attack agents" who launch the actual floods. This preliminary scanning and organization step may take days or weeks to accomplish. Once the list of compromised systems is organized, the intruder installs the appropriate scripts on the client, handlers, and attack agents. The agent installation step takes approximately three to six seconds for each host. The attack network of 2,200 systems used against the University of Minnesota in August 1999 would have taken around two to four hours to set up [Dit00].

Phase 2 of a DDoS attack is the actual denial-of-service attack. The mastermind intruder communicates with the client when he or she wants to instigate an attack at a certain moment. More often, in order to conceal his or her involvement in the attack, the mastermind intruder installs certain instructions on the client that specify the date, time, duration of the attack and the intended victim. Based on the attack blueprint, the client automatically commands the handlers to launch an attack against the intended victim at the specified date and time. Upon receiving the client's command, the handlers coordinate the attack agents to start a traditional DoS attack against the ultimate victim. The actual attack can be of any DoS types, such as the ping flood bandwidth consumption attack [Dit99]. The overall attack impact is amplified by the number of attack agents involved. A typical DDoS architecture is illustrated in Figure 1.1.

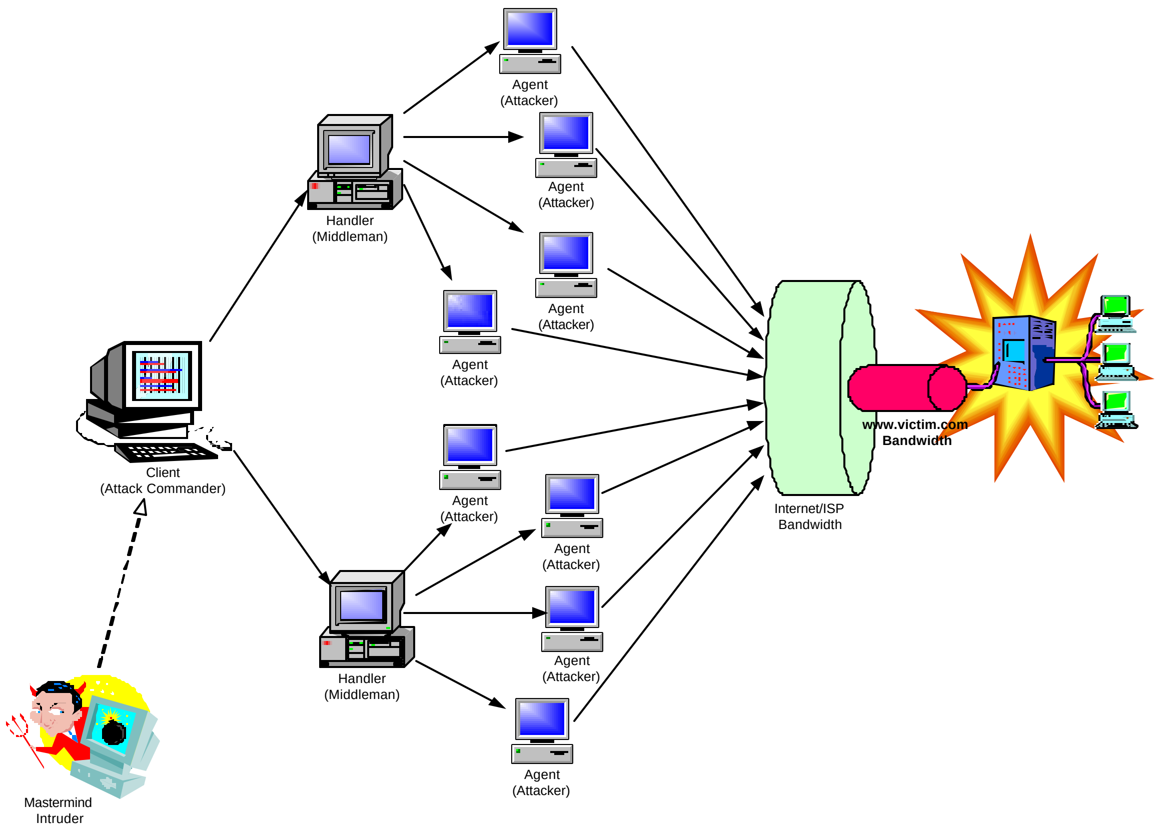


Figure 1.1 – A typical DDoS Architecture

The distributed nature of the DDoS attack has made it extremely difficult to trace and stop the attack. While a single DoS attacker may not be able to shut down a large web-based commercial site such as Amazon.com, the cooperative power of a diverse group of attack agents with various levels of resources can easily make any network inoperable. With one DoS attacker, it is possible to trace and stop the attack. When the attack source composes thousands of computers distributed across the world, the task to trace and stop the DDoS attack instantaneously becomes improbable. In addition, since there is no communication between the mastermind intruder and the client during an attack, it is often impossible to trace back to the mastermind intruder even if the agents, handlers and the client can be identified.

1.3 Mitigation Systems Against DoS and DDoS

To date, a myriad of commercial devices have been introduced to attempt to combat DoS and DDoS attacks. Network Computing has conducted a comparative study on some of the more promising anti-DDoS systems in the Neohaspsis labs in Chicago and the results were released in December 2001[For01]. The anti-DDoS devices studied vary vastly, ranging from switches, routers, to load-balancers, firewalls, intrusion detection system (IDS) and traffic analyzers. Some devices, such as the Mazu TrafficMaster Enforcer and the Reactive FloodGuard, were designed solely for the purpose of handling DoS attacks. Others, such as the Foundry ServerIron 400 and Top Layer AppSwitch 3500, mitigate DDoS through their application switching function. Products such as Captus Networks CaptIO G-2 and Radware FireProof are firewalls that guard against DoS. While these systems provide automatic mitigation, the cost ranges from a monthly charge that starts at $5,000 to a one-time product purchase price of $150,000.

Other devices, such as Asta Networks' Vantage System Enterprise and Arbor Networks' Peakflow DoS, do not respond to DoS attacks automatically. Instead, these devices perform sophisticated data analysis on abnormal traffic patterns and advise the administrators on appropriate mitigation actions. These systems may provide more accurate reports of possible attacks and reduce the chances of automatically blocking legitimate traffic during false alarms. However, like the automated mitigation systems evaluated in the study, the cost is hefty with starting prices ranging from $8,000 to $130,000.

Administrators interested in assembling their anti-DDoS systems do not face cheaper alternatives. An average IDS, such as the Cisco IDS, the Dragon IDS and the ISS RealSecure costs $7,500 to $25,000 depending on the number of nodes and network bandwidth [Des02]. Firewall systems, such as Cisco PIX, Check Point firewall, Nokia IP platform, Sysmaster from SysMaster Corporation, Watchguard Firebox and Sidewinder firewall, range from around $300 to $14,000. The cost of setting up a network against DDoS attacks can quickly sum up to tens of thousands of dollars after adding a few routers and other network components. Despite the financial investment, it is impossible for these devices to defend against all types of DDoS completely due to the changing nature of the attacks.

According to the research conducted by UCSD in 2001, a predominant number of DDoS are targeted towards home networks and to smaller and medium-sized businesses [MVS01]. Since the commercial systems against DDoS are expensive and yet imperfect solutions, small networks may see their needs to guard against DDoS as a low priority and thus increase their chance of being victimized. Other home and medium-sized businesses may not have the resources, knowledge base, and financial means to implement the anti-DDoS commercial systems described above. Therefore, this thesis explores an autonomous defense-architecture against DDoS that can be easily deployed in a small or medium sized network where administrators' time is scarce and financial support is limited. Specifically, the Autonomous Anti-DDoS (A2D2) network proposed in this thesis aims to maximize the quality of service of the victim network automatically during a DDoS bandwidth consumption attack.

Chapter 2

DDoS Defense Research

In general, DDoS defense research can be roughly categorized into three areas: intrusion prevention, intrusion detection, and intrusion response. Intrusion prevention focuses on stopping attacks before attack packets reach the target victim. Intrusion detection explores the various techniques used to detect attack incidents as they occur. Intrusion response research investigates various techniques to handle an attack once the attack is discovered. In addition to these three research areas, intrusion tolerance, once a sub-field of intrusion response, is emerging as a critical research domain. Intrusion tolerance responds to attacks by minimizing the attack impact. This section reviews key research in each of these four areas.

2.1 Intrusion Prevention

The best mitigation strategy against any attack is if the attack never occurs. Research in intrusion prevention has stressed the importance of a well-defined security policy. Another approach to prevent DDoS attack is to stop the attack traffic before it reaches the victim as presented by research on Ingress and Egress filtering.

2.1.1 General Security Policy

"A security policy defines the set of laws, rules, and practices that regulate how an organization implements, manages, protects, and distributes computing resources to achieve security objectives." [CERT98]. Security policies help organizations to define the type of DDoS and other threats they choose to guard against and to incorporate various advisories to prevent these threats.

2.1.2 Ingress and Egress Filtering

Many DDoS tools alter the source IP addresses of attack packets to illegitimate addresses in the Internet name space such as "0.4.7.28". Tracing of the attack source will indicate the illegitimate IP address and the firewall therefore "effectively" blocks a non-existent address as shown in Figure 2.1.

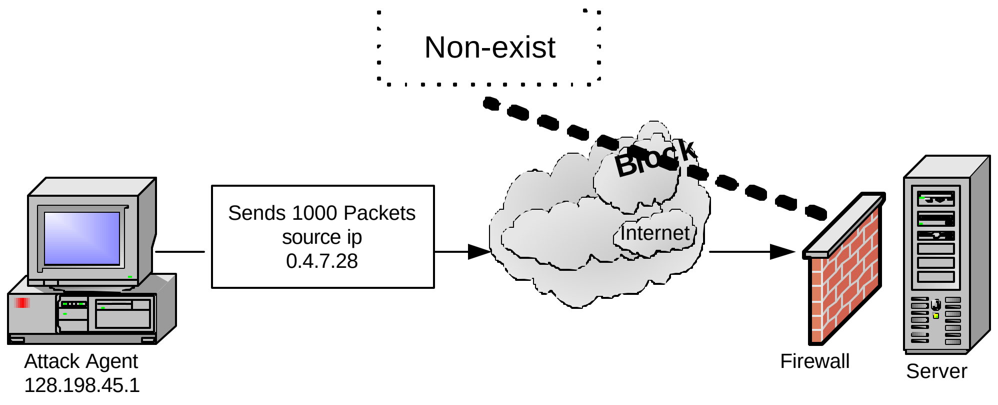


Figure 2.1 - Illegitimate IP Spoofing

Alternatively, the source addresses of the attack packets can be changed to that of a victim’s so that the real victim will be identified as the attacker. By tracing and blocking the source IP of such spoofed packets, the DDoS mitigation network essentially denies service to the real victim and achieves the denial of service that the master attacker intends.

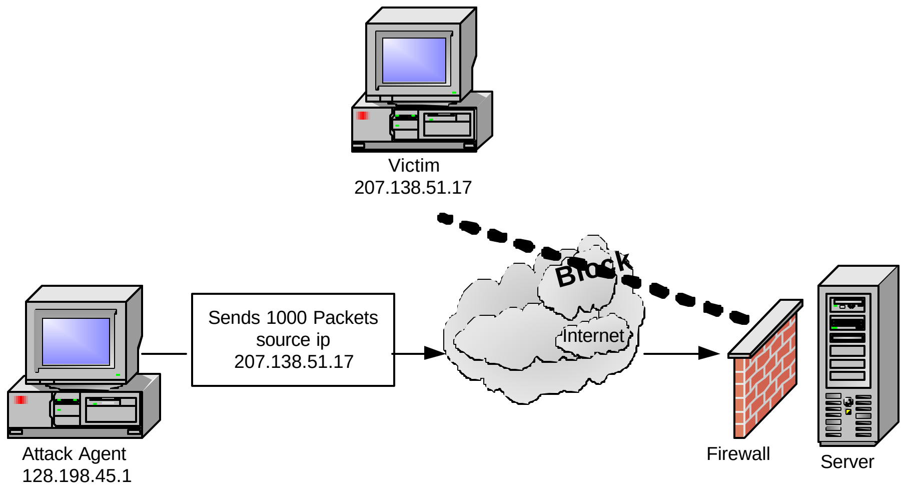


Figure 2.2 – Victim IP Spoofing

IP spoofing has made it impossible to track down the attack source. Research on ingress and egress filtering has targeted this “feature” of the DDoS tool and presented an effective way to limit the damage created by such a DDoS attack [Cis99, FS00]. In egress filtering, network routers will only route packets with addresses from its own assigned IP addresses space to the Internet [San00]. Ingress filtering denies all traffic with addresses nonconforming to the Internet Address space from entering a network. A router implementing ingress filtering can also restrict transit traffic that originates from a downstream network to known prefixes [IET00].

Therefore, ingress and egress filtering can intercept an attack packet with an illegitimate source IP address; discard the packet; and thereby prevent the attack packet from reaching the Internet and the victim network. If an attack originates from a legitimate network, ingress filtering will not be able to filter out attack packets. However, since attackers are forced to use addresses from real networks, filtering can, at the minimum, expedite packet tracing and enable quick identification of attack agents.

While egress filtering can be effectively implemented by many companies or organizations, it becomes difficult or almost impossible for major service providers to take advantage of such techniques. These service providers frequently need to forward legitimate traffic that is not part of its own address space. Egress filtering also becomes extremely complicated in situations where alternate routes are used for traffic traversing through various major service providers. Nonetheless, ingress and egress filtering are effective ways to prevent DDoS attacks based on spoofed IP when the techniques are deployed close to the end user community [San00].

2.2 Intrusion Detection

An intrusion detection system (IDS) can be installed on a specific host to detect invasion against the one host, or be positioned in a network where the IDS can monitor all network traffic in a promiscuous mode. Based on the method of detection, both host-based and network-based IDSs can be divided into two categories: anomaly intrusion detection and misuse or rule-based intrusion detection [Kum95, Stal99].

2.2.1 Anomaly Detection

Anomaly detection first quantifies the ‘usual behavior’. System behavior can be login and session activities such as the login frequency by day, time and location, output quantity or session resource utilization. Behavior can also be command or program execution activity such as execution frequency or program resource utilization. Other behavior tracked can be file access activity such as file read, write, create, and delete frequency [Stal99]. In the case of detecting DDoS bandwidth consumption attacks, behavior of interest would be the bandwidth utilization of the server network or the session activities received by the clients.

By observing past history and collecting data of legitimate behavior over a period of time, network administrators can obtain a behavior baseline of normal activities. Statistical science is then applied to help define the threshold level or the profile that differentiates abnormal behavior from normal behavior [Stal99]. The statistical science applied for such differentiation purpose is well developed. Some techniques used are Bayesian statistics, covariance matrices, classifying schemes, machine learning mechanisms and neural networks [Kum95]. The IDS identifies and alerts activities that surpass the thresholds or deviate from the normal behavior profile as attack.

2.2.2 Misuse Detection

Misuse detection identifies well-defined patterns of known exploits and then looks out for the occurrences of such patterns. Intrusion patterns can be any packet features, conditions, arrangements and interrelationships among events that led to a break-in or other misuse. These patterns are defined as intrusion signatures [NCFF01]. In the anti-virus industry, signatures development has been an established field. Anti-virus products provide signature identification capabilities and virus removal functions. Network intrusion signature development, on the other hand, has just emerged as a new research area. Each IDS vendor may have vastly different signature approaches and methodologies. A recommended approach in Network IDS signature development is to focus on the characteristics of the attacks, vulnerabilities and exploits [Fred02]. For example, "ping commands" executed by administrators send out legitimate ICMP packets to checks for network connection. These ICMPs have small sizes around 64 bytes. In order to consume more bandwidth, attackers often send out a large number of ICMP packets that may be as big as 1500 bytes. Based on this attack pattern, an intrusion signature can identify packets whose type is ICMP and the packet size is greater than 800 bytes. Such an “ICMP Large Packet” signature can generate false positives since ICMP packets can be of various sizes up to 1500 bytes. For example, some load balancing applications such as HP-UX systems use 1500 byte ICMP packets to determine the most efficient route to a host by measuring the latency of multiple paths [Whit01]. However an administrator may still choose to activate this signature for his or her network because the incident of legitimate large ICMP packets arriving at his or her network is extremely rare.

2.2.3 Limitations of Anomaly and Misuse Detection

One limitation of anomaly detection is that it is possible for an attacker to “train” an IDS to reclassify the once abnormal behavior as normal behavior. It is also difficult to determine the thresholds above which a behavior may be considered intrusive. In addition, legitimate users are aware of the normal network behavior, as their actions constitute the “normal” behavior pattern. Unauthorized access or usage conducted by a legitimate user is easily hidden within anomaly detection. On the other hand, misuse detection can identify an attack signature whether the attack is conducted by authorized or unauthorized personnel. However, signatures become obsolete once attack patterns change. New attacks can be easily created with slight modifications to old patterns and new signatures are required. In the event of the “ICMP Large Packet” signature described in Section 2.2.2, administrators are required to collect and analyze network data to determine the size of the common ICMP packets the network receives.

Based on this normal pattern, an administrator can then define the signature of what ICMP packet size is considered “large”. Therefore, to effectively detect intrusion, a network IDS needs to combine both the statistical anomaly part to measure aberration of behavior, and a misuse part that monitors the occurrence of specific patterns of events [Kum95, Stal99].

2.3 Intrusion Response

Once an attack is identified, the immediate response is to identify the attack source and block its traffic accordingly. Improving attack source identification techniques can expedite the capture of attackers and deter other attack attempts. In a distributed attack as illustrated in Figure 1.1, attack agents are often disperse around the globe and bear no relation with one another, and handlers and clients are a few hops away from the attack front. All these elements prolong the time taken to trace and identify the real attack source. Before the master intruder can be identified, networks often respond to attacks by limiting the rate of certain incoming traffic or implement other bandwidth management techniques in hope of maintaining its quality of service to legitimate clients. This type of response is referred to as intrusion tolerance. Intrusion tolerance has emerged as an important research topic and will be described in Section 2.4.

2.3.1 Source Identification

Source identification research investigates techniques to efficiently and effectively identify the attack source despite spoofing and distributed tools. Criminal arrests made possible by speedy and accurate source identification can be a hindrance to other would-be attackers. The best-known work in this category includes ITRACE [IET02, BLT02, MMW+02] and DECIDUOUS [CNW+99].

DECIDUOUS – Decentralized Source Identification for Network-Based Intrusions utilizes the IP Security Protocol (IPSEC) header information to trace the attack packet. Authentication Header (AH) of IPSEC comes after the basic IP header and contains cryptographic hashes of the data and identification information, including source and destination information [OBSD02]. If an attack packet successfully passed a certain router, the router must have authenticated the packet and the router information will be contained in the AH of the packet. From an attack victim, a secure traceroute can be performed that iteratively displays the steps an attack packet has taken to the actual router closest to the source, regardless of what source and destination IP address is.

Among traceback techniques, ITRACE – ICMP traceback and reverse ITRACE has been proposed as an industry standard by the Internet Engineering Task Force (IETF) [IET02]. In ITRACE, routers generate a “traceback message” that is sent along with the forwarded packet to the destination, indicating the router identity that the packet just transited. In reverse ITRACE, the traceback message is sent to the packet’s claimed source instead of destination. These traceback messages are logged and analyzed by special hosts in chosen points in the network. With enough traceback messages from enough routers along the path, the traffic source and path can then be determined [IET02].

2.4 Intrusion Tolerance

Intrusion Tolerance research accepts the fact that it is impossible to prevent or stop DDoS completely. Instead of defeating DDoS, research in this category focuses on minimizing attack impact and maximizing the quality of its services. Many advances in intrusion tolerance are developed based on two other disciplines: fault tolerance and quality of service (QoS).

2.4.1 Fault Tolerant

Fault tolerant is a well-developed research area and fault tolerant designs are built-in in most critical infrastructures. There are three levels at which fault tolerance can be applied: hardware, software and system. Hardware fault tolerance represents the traditional fault tolerant measures where extra hardware resources are used to continue operations in an event of hardware faults. Software fault tolerance introduces mechanisms such as checkpoint restart and recovery blocks to compensate for design errors or data structure faults. System fault tolerance compensates failures in other system facilities that are not computer-based [NIST95] and ensures availability of the entire network and system. The fault tolerant principles of redundancy and diversity across the hardware, software, system and network level can be readily leveraged by the intrusion tolerant community [GWW+00]. By duplicating its services and diversifying its access points, a network can find ways to continue its services when one network link is congested by flooding traffic.

2.4.2 Quality of Service (QoS)

Quality of Service (QoS) describes the assurance of the ability of a network to deliver predictable results and services for certain types of applications or traffic [Comp, ZOS00]. Network elements within the scope of QoS include availability (uptime), bandwidth (throughput), latency (delay), and error rate (packet loss rate) [Comp]. Among Internet applications, video and multimedia services that require continuous transmission of high-bandwidth media information are particularly concern with QoS. Among the most standard QoS techniques used to mitigate DDoS are rate-limiting and class-based queuing [Cis02, HMP+01]. These techniques are explained in Section 2.4.2.1. Often, various QoS techniques are integrated to enable a system that demonstrates superior intrusion tolerance and some of these systems are described in Section 2.4.2.2.

2.4.2.1 Intrusion Tolerant QoS Techniques

Among frameworks to provide Internet QoS, Integrated Service (IntServ) and Differentiated Services (DiffServ) have emerged as the principal architectures [ZOS00]. IntServ utilizes Resource Reservation Protocol (RSVP) to coordinate the resources allocation along the path that a specific traffic flow will pass. The link bandwidth and buffer space are assured for that specific traffic flow. Unlike the per-flow based IntServ, DiffServ is a per-aggregate-class based discrimination framework. DiffServ makes use of the type-of-service (TOS) byte in the IP header and allocates resource based on the TOS of each packet.

Within the two frameworks, queue management and scheduling are the two data operations in router that enables Internet QoS. Queue management controls the length of packet queues by dropping or marking packets. Scheduling determines the order by which packets are being sent. Queuing techniques are employed extensively to combat DDoS attacks. There are many queuing disciplines. The oldest and most widely applied queuing technique is Class-based queuing (CBQ).

Class-based queuing (CBQ) or traffic shaping sets up different traffic queues for different types of packets and for packets of different TOS. A certain amount of outbound bandwidth can then be assigned to each of the queues. For example, a Linux router can limit ICMP traffic to only 5% of the bandwidth link that connects the router to the web server while allowing traffic that targets the multi-media service port of the web server 80% of the available bandwidth. Class-based queuing has shown to maintain QoS during DDoS attack on clusters of web servers [KMW01, WO01].

While queuing or traffic shaping determines the way in which data is sent and manages how the outbound link is utilized, the queuing discipline has no control over the inbound link and how fast packets arrive. Another QoS technique rate limiting or traffic policing applies filters to limit the arrival rate of packets. For example, in response to a ping-flood DDoS attack, a system administrator can configure network routers to accept only 10 ICMP packets per second and discard the rest of the incoming ICMP packets. Rate limiting happens at a very early stage of kernel processing and therefore consumes minimum CPU power in dropping packets.

Often, various QoS techniques are integrated to enable a system that demonstrates superior intrusion tolerance. However, implementation of multiple techniques requires significant administrative effort. For example, if multimedia traffic is allotted 80% of the bandwidth, all traffic targeted to the multimedia port will access the assigned bandwidth regardless of its source. Should attack traffic exhaust the assigned CBQ bandwidth, the system administrator needs to manually identify and rate limit possible attack sources. Once rate-limiting is applied for a certain source IP or certain type of packet, the rate limit will be in effect until system administrators manually remove the rate-limiting condition. If the intrusion detection system tends to produce a high frequency of false positives, the quality of service experienced by legitimate clients will likely be degraded for a long period of time due to rate-limiting.

To alleviate administrators' workload and to minimize mitigation response time during an attack, an autonomous system-approach is necessary. Numerous QoS response techniques need to be integrated with detection mechanism so that the intrusion tolerant system can produce an automatic response during an attack with minimum human intervention.

2.4.2.2 Intrusion Tolerant QoS Systems

Various autonomous architectures have been proposed that demonstrated intrusion tolerant during DDoS bandwidth consumption attacks. Some representative systems are the XenoService [YEA00], the pushback mechanisms proposed by

Ioannidis and Bellovin [IB02], and the autonomic response architecture supported by The Defense Advanced Research Projects Agency (DARPA) [SDW+01].

The XenoService [YEA00] proposed an infrastructure of a distributed network of web hosts that respond to an attack on any one web site by replicating the web site rapidly and widely among XenoService servers, thereby allowing the attacked site to acquire more network connectivity to absorb a packet flood. In order to achieve dynamic replication, the Xeno infrastructure requires ISPs worldwide to install Xenoservers, which run on top of Nemesis, an operating system designed to support QoS. These ISPs then offer web-hosting service at a premium price. During an attack, the web site is then replicated to other Xenoservers among the subscribing Xeno-ISPs. While such infrastructure can ensure QoS during DDoS attacks, it is doubtful that a large number of ISPs worldwide will adopt such infrastructure quickly. Small and medium size businesses may not be willing to subscribe to such expensive services.

The pushback architecture is a promising mitigation technique where routers instruct their upstream routers to rate limit during attacks [IB02]. While it is beneficial for a particular network to implement pushback within its own network boundary, the pushback techniques real value can only be realized when ISPs worldwide make agreements on how to honor pushback requests.

DARPA has supported research on sophisticated autonomic response systems based on the Cooperative Intrusion Traceback and Response Architecture (CITRA) and the Intruder Detection and Isolation Protocol (IDIP) [SDW+01]. IDIP is a special protocol for reporting intrusions and coordinating attack trace-back and response actions among network devices. CITRA refers to the architecture of network communities and network components that use IDIP. The CITRA network components can be IDSs, firewalls, routers, or any devices that adopts IDIP to cooperatively trace and block network intrusions as close to their source as possible [SDW+01]. Special communication protocols such as IDIP and the CITRA infrastructure are gaining acceptance but there is currently no standard in such protocol development. The specification of IDIP is not available to the public domain.

While it is necessary to design a system-approach where QoS techniques such as rate limiting and CBQ can be autonomously deployed, many current autonomous architectures require expensive infrastructure investment, extensive cooperation of different entities, or the adoption of a new protocol. A small business owner does not have influence over the network design or the partnerships of his or her service provider. Therefore, the current thesis aims to design an Autonomous Anti-DDoS (A2D2) network by integrating and improving existing methodologies that enable small and medium-sized networks to take control of their own defense within their own boundary.

Chapter 3

The Proposed Autonomous Anti-DDoS Network (A2D2) Design

This thesis proposes an autonomous anti-DDoS network design that utilizes existing and affordable tools and technologies. The goal of the design is to combine various technologies and make necessary improvements to achieve autonomous attack mitigation similar to that attained by elaborate expensive architectures. The A2D2 network is specifically designed to enhance quality of service during bandwidth consumption DDoS attack. The A2D2 design follows four main guiding principles:

- Affordable
- Manageable
- Configurable
- Portable

The target audience for the A2D2 network is home networks and small to medium sized companies. To ensure affordability, A2D2 will make use of open source and existing technologies wherever possible. In addition, the A2D2 network should be easily managed with minimum administrator intervention, can be quickly configured for networks of various sizes and readily ported to mitigate attacks other than DDoS.

The design of the A2D2 network will be divided into three main areas:

- Intrusion Detection
- Intrusion Response: Intrusion Tolerance – Quality of Service
- Autonomy System

3.1 Intrusion Detection

3.1.1 Snort Overview

As mentioned in Section 1.3, well-known systems such as Cisco Secure IDS, the Dragon IDS, ISS RealSecure and Symantec NetProwler cost $7,500 to $25,000 depending on the number of nodes and network bandwidth [Des02, Sao02]. Except for the Windows-based NetProwler, all other IDSes mentioned support a variety of platforms such as Sun Solaris, Linux, and the Windows systems. Among all the well recognized and broadly deployed IDSes, Snort is the only free, open source lightweight intrusion detection system and is selected to be the detection component of

A2D2 [Sao02]. Snort's network monitoring mechanism is based on the pcap packet capture library, which makes Snort's code portable among various platforms that support libpcap. Currently Snort is run on Linux, Net/Open/FreeBSD, Solaris, SunOS 4.1, HP-UX, AIX, IRIX, Tru64, MacOS X Server and the Win9x/NT/2000 platform [Snort].

Snort can be operated in three modes: a straight packet sniffer similar to tcpdump, a packet logger, or as a full-blown network intrusion detection system. As an IDS, Snort performs real-time protocol analysis, content searching and matching, and real-time alert. Attack detection is mainly based on a signature recognition detection engine as well as a modular plugin architecture for more sophisticated behavior analysis.

3.1.2 Snort Detection Engine

The Snort detection engine is based on signature recognition techniques. Signatures are classified into different types such as DoS type or ICMP type and then defined in the rule files of the specific classification such as ddos.rules or icmp.rules. Content searching and matching functions are defined within the detection engine processors. The processors test an aspect of the current packet and report the findings. These functions are accessed from the rules file as standard rule options and may be called many times per packet with different arguments. Rules are checked in sequence according to the order the rules are specified in the rule file.

Using a flexible rules language, Snort enables administrators to describe the type of pattern or traffic that the IDS should pay attention to. The rule, or signature description, can be based on packet source, destination, port numbers, tcp flags, packet size and other header information. Snort also decodes the application layer of a packet and signatures can be designed based on the contents of the packet payload. Snort provides a large library of rules that detect a variety of hostile activities, including buffer overflows, CGI attacks, SMB probes, or any other data in the packet payload that can be characterized as a unique attack fingerprint. Rules for new exploits are readily available on the Snort website. The Snort website also provides detailed documentations and manuals to guide administrators in creating rules that suit their specific security policy [Snort]. A sample Snort rule is:

- alert icmp any any -> 10.1.1.0/24 any (msg:"Being Pinged"; itype: 8;)
    - # alert is the rule action. There are 5 types of actions that can be performed when a certain rule is matched: alert, log, pass (ignore), activate and dynamic. The action “activate” raises an alert and also switches on the specified dynamic rule that in turn logs the event.
    - # the second field of a Snort rule specifies the type of protocol. In this rule, an alert will be generated when an ICMP packet that matches the rule condition arrives at the network.
    - # the third field of a Snort rule describes the source IP address. In this sample rule, “any” means to alert all ICMP packets initiated from any addresses.

- # the fourth field describes the source port. In this example, the rule looks for ICMP packets with “any” source port
- # the destination IP is the subnet 10.1.1.0/24
- # the destination port is “any”
- # alert message is “Being Pinged”
- # itype:8 checks the type field of the ICMP packet header to see if the field contains the number 8, indicating the ICMP packet is a “echo request” packet.

3.1.3 Snort Module Plugin - Preprocessors

In addition to the detection engine, Snort provides a modular plugin architecture that enables more complex analysis on collective packet behavior and performs sophisticated decoding of packet contents. Module plugin preprocessors are not accessed through rules. Instead, raw packets are submitted to the various preprocessors sequentially. Each preprocessor performs some functions once for each packet, then evaluates the condition and alerts if a suspected attack behavior is observed. A preprocessor does not modify the packet information. Packets go through the preprocessors before being passed to the detection engine and being matched against Snort rules.

Preprocessors are important since many attacks cannot be detected merely by matching a simple pattern in a packet header or payload. For example, in a fragmentation attack, attackers take advantage of the fact that the IP protocol allows an IP packet to be broken apart into several smaller packets during transmission and then reassembled at the final destination. The maximum allowable size for an IP packet is 65,536 bytes. Through fragmentation, IP traffic can be transmitted across networks with different maximum packet sizes without being restricted by a particular network in the route that defines a very small maximum packet size. In a fragmentation attack, an attacker may send out a packet whose fragmented sections add up to more than 65,536 bytes. Many operating systems did not know what to do when they received an oversized IP packet, so they froze, crashed, or rebooted [Whatis]. With preprocessors, Snort is able to perform IP defragmentation and then determine if a fragmentation attack is launched. Preprocessors can also perform full TCP stream reassembly and then carry out stateful inspection of the TCP streams to detect intrusions such as portscan types and fingerprinting where attackers secretly attempt to learn about your systems and potential vulnerabilities.

In some HTTP-based attack, attackers may wish to access a specific directory in a URL but do not wish to be detected by the IDS. Instead of naming the directory path in the URL, attackers uses hex encodings to represent certain characters in URLs. For example, an attacker may want to access the directory “scripts/../../winnt”. Instead of using the path “scripts/../../winnt” in the HTTP request, an attacker substitutes the “/” with “%5c”. Since “%5c” is the hex encoding equivalent of a backslash, the URL then contains “scripts/..\../winnt” which is treated the same as “scripts/../../winnt” by most web servers [Fred02]. Snort preprocessors can decode HTTP requests by converting non-ASCII %xx character substitutions to their ASCII equivalent so that attackers who stealthily gain access to systems by mixing these substitutions into URL can be detected.

The preprocessor module plugin architecture also provides a channel through which new types of detection engines can be added. New engines may perform detections that cannot be supported by a Snort base detection engine. For example, Spade, the Statistical Packet Anomaly Detection Engine, is added to Snort through the preprocessor architecture to enable anomaly detection [Snort].

3.1.4 A2D2 Snort Module Plugin – Flood Preprocessor

At present, Snort has not included a logic that detects generic bandwidth consumption flooding launched against a network. DoS and DDoS detections are carried out by the rule files and the base detection engine. For example, a potential Stacheldraht DDoS detection is based on two signatures that match message strings contained in communication messages sent between attack agents and their handler. The agents send messages to inform the handler that the agent machine is alive and ready to take orders. This communication message contains the word “skillz”. In turn, the handler commands the agents to launch attack requests with a message that contains the string "ficken". Two Snort rules are created to detect the presence of “ficken” or “skillz” in a packet payload. This example illustrates a major limitation of pattern and signature base detection. Attackers can easily change the payload content of agent-handler communication messages and new rules will need to be added.

To reduce management and maintenance hassle, A2D2 is required to detect generic flooding attack independent of specific DDoS tools. Unlike pattern or signature matching, flood detection needs to be designed as a preprocessor modular plugin. The flood preprocessor will perform an “x packets over y time” logic evaluation. If x packets arrive within y seconds from the attack source, an attack alarm will be raised. Administrators or users can set an incoming packet rate threshold (x packet over y time) that deviates from their normal network traffic significantly. This flood threshold is set in the snort.conf file and provides a flexible configuration channel compatible with existing preprocessors of Snort.

3.1.4.1 Flood Threshold

Flood threshold differs depending on the type of services provided from the network, the nature of the company, the size of the network, and the time of the day. The flood threshold has to be determined at each network independently. The A2D2 design allows administrator to configure the threshold to be reflective of his or her network traffic. Before a threshold is determined, administrators should collect average bandwidth usage over a period of time. This baseline evaluation should be conducted over a period of months at the minimum to take into account usage surges during specific hours in a day or specific occasions. As mentioned in Section 2.2.1 Anomaly Detection, techniques such as Bayesian statistics, covariance matrices and neural networks help to define the “normal” traffic from the “abnormal” traffic”[Kum95]. These techniques are studied in the research domain of anomaly detection and are beyond the scope of this thesis. In A2D2, the threshold will be determined by doubling the baseline average traffic information.

3.1.4.2 Flood Preprocessor Initiation

Snort has specific directions as to how new preprocessor plugin modules can be incorporated. The A2D2 generic flood detection preprocessor is named spp_flood.c and is accompanied by the spp_flood.h header file. The followings describes the initiation steps required to add the spp_flood preprocessor to Snort:

1. Add to the snort plugbase.h file

   ```
   #include "spp_flood.h"
   ```

2. Add the following lines to the snort plugbase.c file

   ```
   void InitPreprocessor()
    {
           SetupFlood();
    }
   ```

3. Add the following lines to the snort.conf file

   ```
   preprocessor flood: $HOME_NET <threshold # packets> <threshold #
   time period> <logfilename>
   ```

4. Create two flood-plugin files:
   - spp_flood.h
   - spp_flood.c

5. In spp_flood.h, add

```
void SetupFlood();

void FloodInit(u_char *);

# The FloodInit function creates the preprocessor data structure
```

6. In spp_flood.c, register the preprocessors by adding the following function:

```
void SetupFlood(void)
{
        RegisterPreprocessor("flood", FloodInit);
}
```

3.1.4.3 Flood Preprocessor Data Structure

The flood preprocessor floodList maintains the packet rate utilizing a three-dimensional double-linked list:

- floodList → sourceInfo (match source ip)
- destinationInfo (match destination ip)
- connectionInfo (match port info)

The first level list sourceInfo registers the packet source address. For each source, the packet's destination is recorded and counted in destinationInfo. For each source-destination connection, the packet's port information is recorded and incremented. Key data structures used for flood detection are presented below:

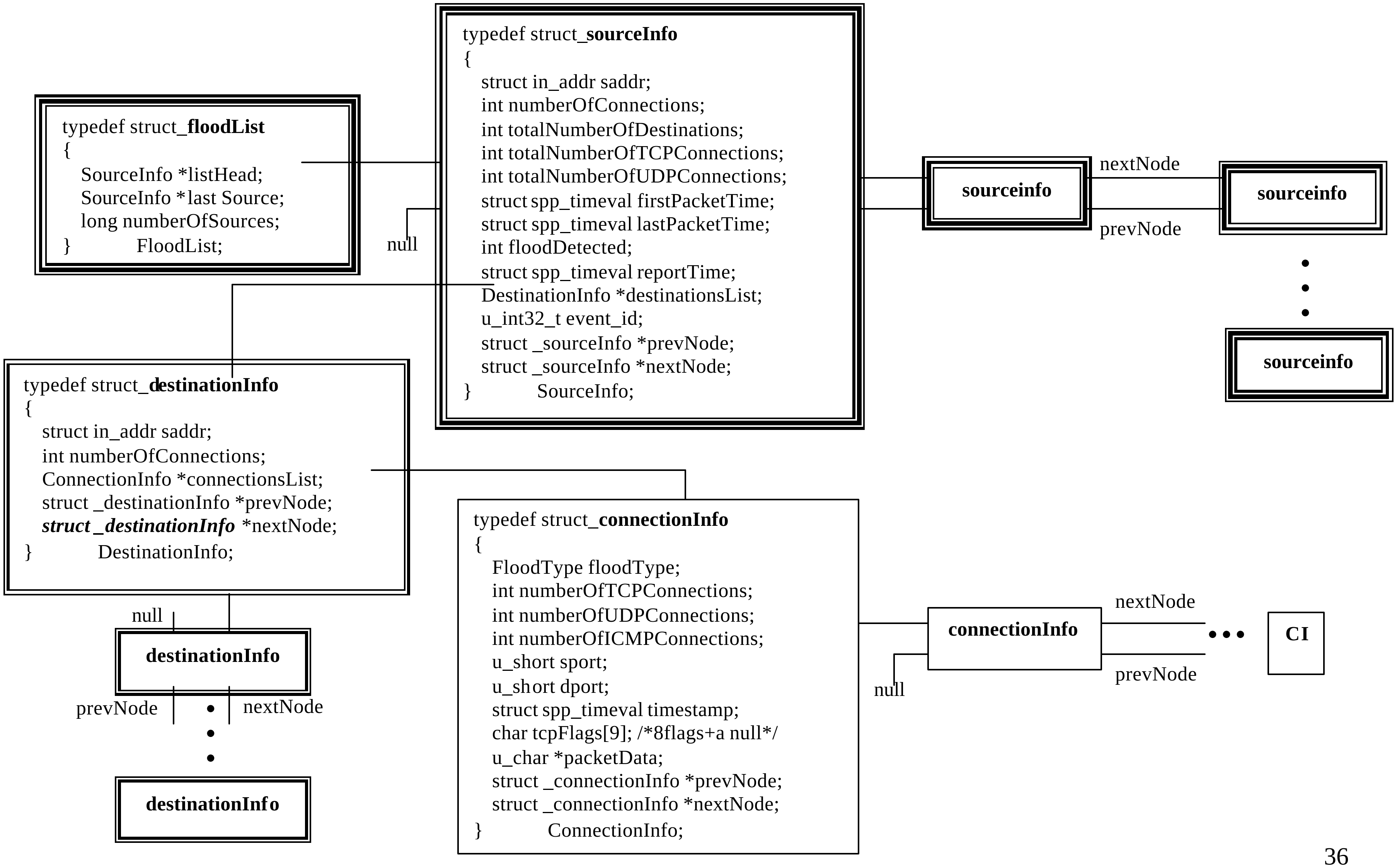
```
struct spp_timeval
{
   time_t tv_sec;
   time_t tv_usec;
};
```

```
typedef enum_ floodType
{
   sNone = 0,
   sUDP = 1,
   sSYN = 2,
   sSYNACK = 4,
   sICMP = 8
}          FloodType;
```

```
typedef enum_log level
{
   lNone = 0,
   lFILE = 1,
   lEXTENDED = 2,
   lPACKET = 4,
}          LogLevel;
```

Figure 3.1 - Flood Preprocessor Key Data Structure

(Figure Continues on Next Page)



```
typedef struct_floodList
{
   SourceInfo *listHead;
   SourceInfo *last Source;
   long numberOfSources;
}          FloodList;
```

```
typedef struct_sourceInfo
{
   struct in_addr saddr;
   int numberOfConnections;
   int totalNumberOfDestinations;
   int totalNumberOfTCPConnections;
   int totalNumberOfUDPConnections;
   struct spp_timeval firstPacketTime;
   struct spp_timeval lastPacketTime;
   int floodDetected;
   struct spp_timeval reportTime;
   DestinationInfo *destinationsList;
   u_int32_t event_id;
   struct _sourceInfo *prevNode;
   struct _sourceInfo *nextNode;
}          SourceInfo;
```

```
typedef struct_destinationInfo
{
   struct in_addr saddr;
   int numberOfConnections;
   ConnectionInfo *connectionsList;
   struct _destinationInfo *prevNode;
   struct _destinationInfo *nextNode;
}          DestinationInfo;
```

```
typedef struct_connectionInfo
{
   FloodType floodType;
   int numberOfTCPConnections;
   int numberOfUDPConnections;
   int numberOfICMPConnections;
   u_short sport;
   u_short dport;
   struct spp_timeval timestamp;
   char tcpFlags[9]; /*8flags+a null*/
   u_char *packetData;
   struct _connectionInfo *prevNode;
   struct _connectionInfo *nextNode;
}          ConnectionInfo;
```

3.1.4.4 Subnet Flood Detection

This version of the flood preprocessor detects only floods launched in ICMP, UDP, TCP-SYN or TCP-SYN-ACK packets. These types of packets are termed "relevant packets" in this document. The flood preprocessor is not concerned with the packet contents and the packet payload will not be checked. The most basic premise of flood detection is that if the number of relevant packets from a particular source within a certain time exceeds the threshold specified in the configuration file, a flood alert is raised. Such logic is effective in detecting traditional floods where one attacker instigates the flooding from one machine with one source IP address. A simple ping flood attack using the command "ping –f <victim domain name or IP>" can be easily detected.

Nowadays, almost all bandwidth consumption DDoS attackers spoof the source IP addresses of the attack machines. As mentioned in Chapter 3, widespread practice of ingress and egress filtering has effectively prevented spoofing of illegitimate IP sources or of addresses of the victim domain. Spoofing is limited to those addresses that reside within the same subnets of the attacker so that attack packets can pass through ingress and egress filtering. To make an attack more efficient, a DDoS attack agent can send attack packets with an array of randomly generated source addresses, all of them within the subnet of the attack agent. Each spoofed address is used in a limited number of packets to reduce suspicion. These spoofed DDoS attacks are illustrated in the figure below:

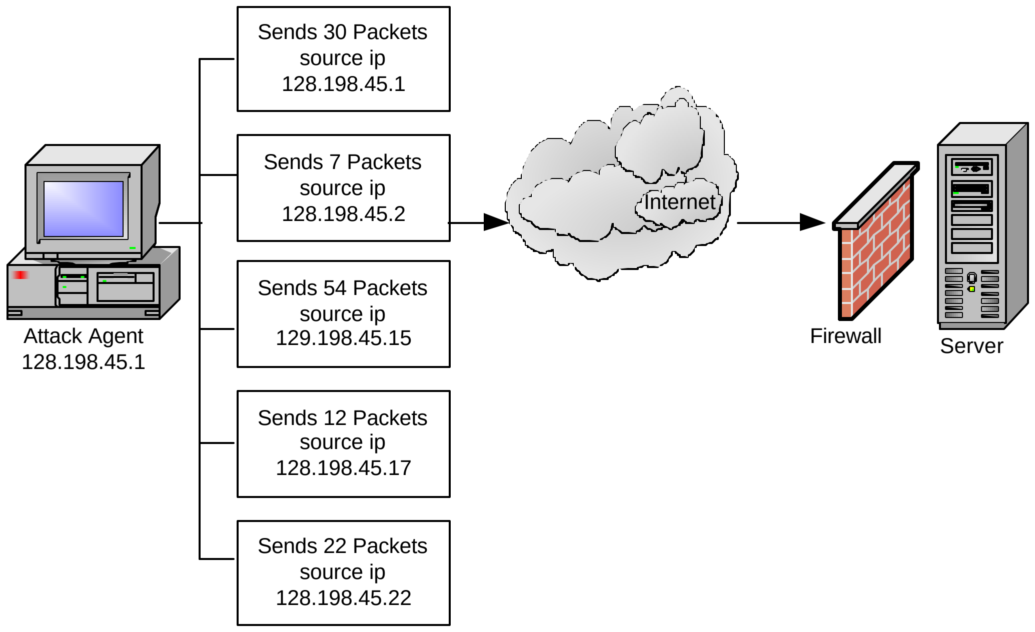


Figure 3.2 - DDoS IP Spoofing Each Attack Agent

To counter DDoS IP Spoofing, A2D2 is designed to detect subnet flooding as well as individual host flooding. The three types of generic flooding that are being detected are:

- Individual attack host against individual victim host
- Subnet attack agents against individual victim host
- Subnet attack agents against victim subnet hosts

With current technology, it is still impossible to identify from which subnet a packet initiated. Therefore, certain design assumptions have been made regarding subnet flooding detection. For subnet flood detection, A2D2 will assume packets come from a /24 network based on the Classless Inter-Domain Routing (CIDR) addressing scheme [RL93]. A /24 network is equivalent to a traditional Class C network with 253 host addresses and three other addresses for network, broadcast and gateway identification. In this thesis, it is assumed that most attack tools will not forge source IP beyond the realm of the /24 network in order to ensure that attack packets will pass through ingress and egress filtering.

Considerations have been given to /22 and /16 subnet flood detection. There are 1021 hosts in a /22 network and 65,533 hosts in a /16 network respectively. These networks can legitimately generate a large amount of traffic. A /22 and /16 subnet flood detection adds extra reassurance but may also produce more false positives. If the flood threshold is set high to accommodate possible simultaneous connections from all 65,533 hosts in a /16 subnet, the IDS may risk a large number of false negatives. In practice, most existing networks are partitioned in smaller subnets with less than 253 hosts such as /25, /26, /27 networks. Therefore, the A2D2 design will assume a /24 subnet flood detection. Another assumption made in the design of A2D2 subnet detection is that many networks are implementing ingress and egress filtering as described in Section 2.1.2 so that most spoof packets beyond the IP addresses of a /24 subnet will be discarded.

With the assumption of a /24 subnet flood possibility, it is recommended that the flood threshold be set at a level that accommodates reasonable connections from all hosts in a /24 network. For example, an Internet Control Message Protocol (ICMP) is used mainly by administrators or applications to determine the state of a particular server; it is highly unlikely that all 253 hosts from a subnet will send an ICMP packet to the server simultaneously. In a situation where all hosts are connecting to the server at one time, the number of reasonable packets should still remain within 256 hosts X 2 packets (syn, syn-ack) = 512 packets within a 3-5 second period. As mentioned before, the actual threshold should be set only after baselining the general network traffic pattern.

3.1.4.5 Flood Preprocessor Logic Flow

Complete features and functions of the A2D2 Snort flood preprocessor will not be described in detail in this document. Instead, key functions of the spp_flood.c module are abstracted to illustrate the flood detection logic. These functions are:

- void FloodPreprocFunction(Packet *p)
- void ExpireFloodConnections(FloodList * floodlist, struct spp_timeval watchPeriod, struct spp_timeval currentTime)
- int NewFlood(FloodList * floodList, Packet * p, FloodType floodType)
- int CheckSubnetIndFlood(FloodList * floodList, Packet * p, FloodType floodType, int *subnet)
- int CheckSubnetFlood(FloodList * floodList, Packet * p, FloodType floodType, int *subnet)

The main function of the flood preprocessor, void FloodPreprocFunction (Packet *p), controls the logic flows and other key function calls. The flowchart depicting the logic is provided in Appendix A. The FloodPreprocFunction performs the following major steps:

1. Packet checking & preparations

    - The FloodPreprocFunction first determines the protocol, flags set for a TCP packet, timestamp and other relevant information of the incoming packet. If the packet is not a relevant packet, the function exits the module.

2. Expire non-flood connections in the data structure

    - The function **void ExpireFloodConnections** (FloodList * floodlist, struct spp_timeval watchPeriod, struct spp_timeval currentTime) is called.

        - Check the "flood detection" flag and the time stamp of a connection.
        - If currentSource is flagged as floodDetected, do not expire.
        - If currentSource is not flagged as floodDetected, check time stamp.
        - If (connection's timestamp + watchPeriodTime is < currentTime), the function expires and removes the connection from the data structure.

3. Log current packet in the data structure

- The function **int NewFlood**(FloodList * floodList, Packe * p, FloodType floodType) is called.
  - Traverse through the data structure to see if the current packet matches a certain connection base on its source IP address, destination IP address and port information. If a match is found, increment existing connection information or packet counts of a certain source and destination. If a match is not found, create a new source or destination connection
  - The int NewFlood function returns the maximum number of packet counts of a source-destination connection in the data structure.

4. Check flood: individual attack host against individual victim host
   - Check if the maximum number of packet counts in a source-destination connection in the data structure has passed the flood threshold.
   - Log, alert, and flag connections as flood detected as appropriate.
   - If individual flood is determined, subnet flood will not be checked.
5. Check subnet flood: subnet attack agents against individual victim host
   - If individual flood is not found, the function **int CheckSubnetIndFlood**(FloodList * floodList, Packet * p, FloodType floodType, int *subnet) is called.

   - Traverse through the data structure and count all the connections that could have come from the same subnet as the current packet, assuming a /24 subnet mask.
   - The int CheckSubnetIndFlood function returns the maximum number of packets from assumed connections in the /24 subnets.
- Check if the maximum packet counts has exceeded the flood threshold.
- Log, alert, and flag connections as flood detected as appropriate.

6. Check subnet flood: subnet attack agents against victim subnet hosts
   - If no other flood is detected, the function **int CheckSubnetFlood**(FloodList * floodList, Packet * p, FloodType floodType, int *subnet) is called.
     - Count all the connections whose destination are from the same subnet as the current packet's destination (assume /24 subnet)
     - Return the maximum number of packets from the current packet subnet source targeted to destination addresses of the same subnet.
   - Check if the maximum number of packet counts from the specific subnet-subnet connection has exceeded the threshold

- Log, alert, and flag connections as flood detected as appropriate

7. Check if the flood has ended and clear the floodDetected flag accordingly
   - Go through the data structure and evaluate the “floodDetected” connections
   - Check if the floodReportTime + maxTime < currentTime
     - If the floodDetected source IP have 0 new connection during that period, clear the floodDetected flag.
     - Alert and log “End of Flood” as appropriate

3.1.5 A2D2 Snort Module Plugin Add-on – Flood IgnoreHosts Preprocessor

It is common that administrators may perform bandwidth measurement tasks or other administrative and diagnostic functions that require sending out a large number of packets to the network and an IDS may identify such activities as floods. In other cases, a certain service may be provided to a valued-customer at a special occasion that will generate significantly more than “normal” connection request traffic from a large /16 subnet with 65,536 hosts simultaneously. To accommodate such situations, A2D2 IDS detection includes another preprocessor add-on FloodIgnoreHosts. The FloodIgnoreHosts preprocessor is added to the base spp_flood.c preprocessor so that Snort will ignore counting the number of packets generated from a particular server IP or a particular network. The setup of the FloodIgnoreHosts preprocessor is similar to the setup of the flood preprocessor.

1. Add the following in the snort plugbase.c file

```
void InitPreprocessor()
 {
          SetupFloodIgnoreHosts();
 }
```

2. Add the following lines to the snort.conf file

```
preprocessor flood-ignorehosts: <ip_hoststobeignored/subnetmask>
```

3. In spp_flood.h, add

```
void SetupFloodIgnoreHosts(void)
void FloodIgnoreHostsInit(u_char *)
# The FloodIgnoreHostsInit function creates the preprocessor data
structure
```

4. In spp_flood.c, register the preprocessors by adding the following function:

```
void SetupFloodIgnoreHosts(void)
{
          RegisterPreprocessor("flood-ignorehosts",
FloodIgnoreHostsInit);
}
```

A separate data structure is created to keep track of what servers or subnets the IDS should ignore:

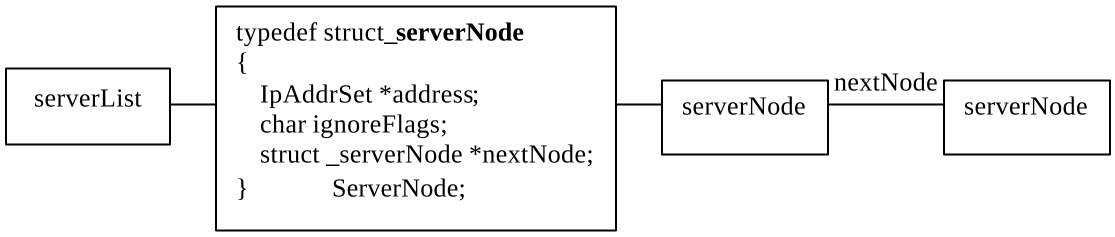


Figure 3.3 – FloodIgnoreHosts Preprocessor Add-on Key Data Structure

The IgnoreFloodHost function int IsFloodServer(Packet *p) is called immediately after a packet is checked and prepared in void FloodPreprocFunction. The function returns 1 if the source address of the packet is in the "flood-ignored" serverList and returns 0 otherwise. If a 1 is returned, the FloodPreprocFunction is exited and the subsequent incoming packet will be passed to FloodPreprocFucntion for processing.

3.2 Intrusion Responses – QoS

The mitigation strategy adopted by A2D2 is based on QoS research. As mentioned in Section 2.4.2, Class-Based Queuing (CBQ) and rate limiting are two dominant QoS techniques. While setting up a solid security policy is considered a preventive measure rather than a QoS mitigation, a security policy is a critical aspect of any network. The A2D2 network design policy will described briefly in the following section.

3.2.1 Security Policy

The principal security policy applied is the separation of public services from the private network. Indeed, the design of A2D2 centers on the design of the anti-DDoS Demilitarized Zone (DMZ). A DMZ is a small network inserted as a "neutral zone" between a company's private network and the outside public network [Whatis]. Users of the public network outside the company can access only the hosts in DMZ. All public services available freely to the world, such as web services and file services with insensitive data, reside in DMZ servers. DMZ servers cannot initiate sessions into the private network. In the event that an outside user penetrated the DMZ hosts' security, only the web pages and other public information might be corrupted but no other company information would be exposed. A typical DMZ and its IDS placement are illustrated below:

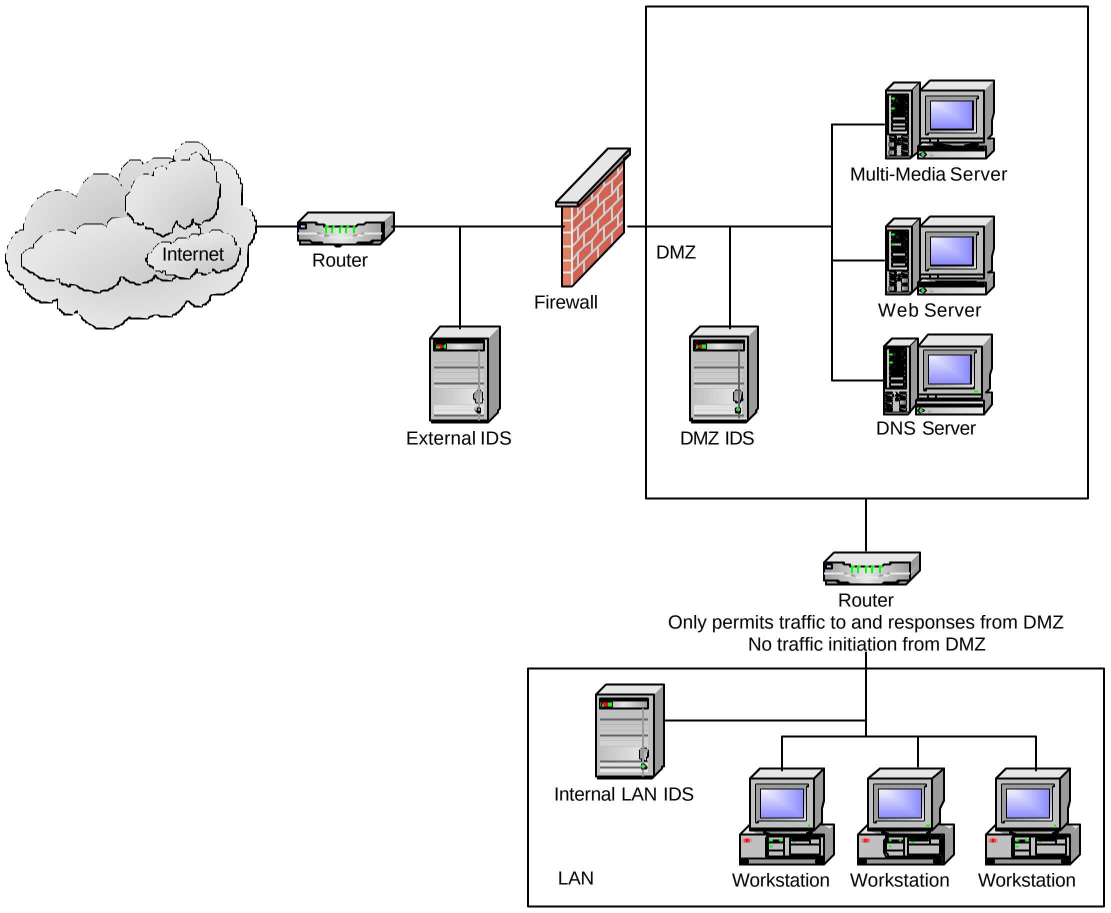


Figure 3.4 – A Typical DMZ and its IDS Placement

An external IDS is placed outside the firewall before any traffic enters the DMZ. The IDS should be configured to be least sensitive and will produce the most false alarms. It provides an overview for administrators to evaluate all traffic attempts to access the network. Administrators can use the external IDS log to shape long-term network security policy. An IDS is placed within the DMZ to catch possible intrusion that penetrated the firewall. The DMZ IDS produces less false positives and can reflect the effectiveness of the firewall. Depending on the type of intrusions detected by DMZ IDS, alarms should be treated with caution promptly. Any traffic that reaches the trusted internal LAN is considered hostile unless authorized. The LAN IDS generates the least number of false positives and any detection alarm demands immediate action [Cis02-2, CR00, Lai00, San00].

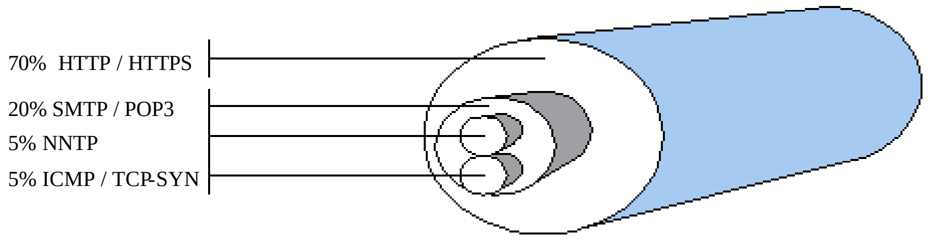
The firewall implements a set of rules or chains based on the network security policy. A recommended approach is to set a deny policy where all traffic are denied into the DMZ. Based on the services started in the DMZ public servers, additional rules are applied to allow traffic to access the specific port. Examples are:

- /sbin/iptables – P INPUT DROP
- /sbin/iptables – P OUTPUT DROP
- /sbin/iptables – P FORWARD DROP
- /sbin/iptables – A INPUT – p tcp – – dport 80 – j ACCEPT

The above example sets the default policy (-P) for the INPUT, OUTPU, and FORWARD chains to be "DROP", meaning all packets will be discarded. The INPUT chain defines rules that will be applied to packets targeted for the specific computer on which the policy is set. The OUTPUT chain sets rules for packets that are sent out by the computer where the policy is set. The FORWARD chain contains rules that govern what actions will be carried out for packets that are passed through or routed by the policy computer. The last rule of the above example appends a rule (-A) to the INPUT chain to "ACCEPT" TCP packets (-p tcp) that are targeted to the destination port 80 (- - dport 80).

3.2.2 Class-based Queuing (CBQ)

As mentioned in Section 2.4.2.1, CBQ is an effective QoS technique for controlling traditional SYN and ICMP floods. Based on the user access policy, a certain percentage of available outbound bandwidth can be assigned to packets of various Types of Services (TOS). The concept of CBQ can be best illustrated by Figure 3.5. Assume that an administrator of a certain network analyzes the traffic pattern and decides that HTTP traffic should be guaranteed at least 70% of the available bandwidth while mail services be allowed 20% of the network link. The administrator then divides the remaining bandwidth between news services (5%), and TCP-SYN and ICMP traffic (5%).



(Graphics adapted from Figure 2, "Linux Firewall – the Traffic Shaper" by Wortelboer and Oorschot [WO01])

Figure 3.5 - Bandwidth Assignment Example by CBQ

Inside the Linux kernel, the bandwidth management is achieved by three components: the filter/classifier, the queues, and the scheduler [AV02, HMM+02]. As a packet arrives at the network interface, the kernel discards the packet, forwards the packet or marks it as a certain class to be passed on to the queuing disciplines such as CBQ or First-in-first-out (FIFO). If the queuing discipline supports classes, the queuing discipline in turn moves the packets to the classes of queues. CBQ supports a maximum of eight separate queues, or classes. Each queue or class can then be assigned a policy that identifies the priority, bandwidth allocation, bounded, or queue size. A bounded queue is constricted by the assigned bandwidth but an unbounded queue can “borrow” bandwidth from another queue with extra bandwidth. CBQ queues in an order that will honor the bandwidth allocation defined by the queuing policies of all the queues. The scheduler visits each queue and examines the maximum bandwidth policies of the queues. If forwarding a packet from a particular queue will violate the bandwidth allocation, the scheduler will skip the packet and move to another queue of the same priority [AV02, HMM^{+}02]. The implementation of CBQ is illustrated in Figure 3.6.

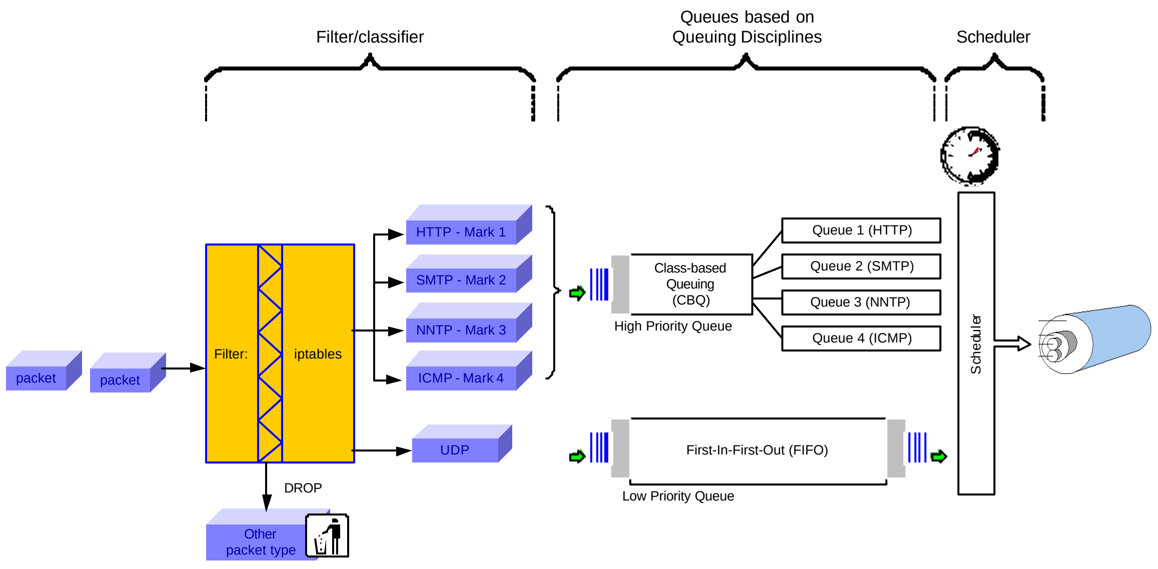


Figure 3.6 - Implementing QoS using Class-Based Queuing

In Linux, CBQ is implemented using the mangle table of the IP filter as well as the traffic control (tc) program from the “iproute2” package located in the directory “sbin” (/sbin/tc). The mangle table enables iptables to mark and categorize packets. The tc program creates the different classes of queues and assigns their policies regarding bandwidth usage. There are 5 steps in the implementation:

1. Enable Linux QoS and CBQ options under the “Networking Options” menu [HMM+02].
    - Change current directory to /usr/src/linux<version>/
    - Type “make menuconfig” at command line
    - Choose “Networking Options”
    - Select “QoS and/or fair queuing”
2. Categorize, mark, and forward incoming packets using iptables [WO01].
    - /sbin/iptables – A FORWARD – p icmp – t mangle – j MARK --set-mark 1
        - Append this rule (-A) to the FORWARD chain of the mangle table (-t). The mangle table allows iptables to change or modify packets before they are routed. In this case, a packet is marked with the number 1 (-j MARK --set-mark 1) for all ICMP packets (-p icmp).

- /sbin/iptables – A FORWARD – p tcp – – syn – t mangle – j MARK -- set-mark 2
  - Use the mangle table to mark all TCP-SYN packets (-p tcp --syn) with the number 2.
- /sbin/iptables – A FORWARD – p tcp --dport 80 – t mangle – j MARK – – set- mark 3
  - Use the mangle table to mark all TCP packets (-p tcp --syn) that target destination port 80 (-p --dport 80) with the number 3.

3. Set up the queue (qdisc) with the specific network interface (add dev eth0). This step defines the queuing discipline of the qdisc (cbq) and the maximum bandwidth associated with such queue. Classes are created under a queue based on a tree structure. While setting up the queue, the root of the tree is named (root handle 10:). The queue also specifies the average size of the packets measured in bytes (avpkt) [AV02, HMM+02].
   - # Assume a 10 Mbit network link
   - /sbin/tc qdisc add dev eth0 root handle 10: cbq bandwidth 10Mbit avpkt 1000
4. After the queue is set up, the root class associated with the queue is created and initialized. The root class is identified (classid 10:1) and it is spawned from the root of the tree (parent 10:0 or parent 10:). The

class priority and the amount of data sent by each class are defined by "prio", "allot" and "weight". Packets will be scheduled for output first by priority and second by allocation. Classes using the same priority will be scheduled using a Weighted Packet Round Robin algorithm (WRR) based on the allocation values and the weight assigned. The weight can be any arbitrary numbers. The weights of all classes will be normalized to calculate the ratio. This ratio will be multiplied by the "allot" parameters to determine how much data will be sent out by the class when polled by the scheduler [AV02, HMM+02].

- /sbin/tc class add dev eth0 parent 10:0 classid 10:1 cbq bandwidth 10Mbit rate 64kbit allot 1514 weight 6.4kbit prio 8 maxburst 20 avpkt 1000 bounded

5. Create different classes of queues with different bandwidth allocation policies and instruct IP filter to send packets to the queue and its appropriate class [WO01]. The various parameters defined in these classes are explained in steps 3 and 4.

   - add_class() {

     # $1=parent class $2=classid $3=hiband $4=lowband $5=handle $6=style

```
    /sbin/tc qdisc add dev eth0 parent $2 cbq bandwidth 10Mbit
    avpkt 1000

    /sbin/tc class add dev eth0 parent $1 classid $2 cbq bandwidth
    10Mbit rate $3 allot 1514 weight $4 prio 5 maxburst 20 avpkt
    1000 $6

    /sbin/tc filter add dev eth0 protocol ip prio 3 handle $5 fw
    classid $2

}
```

- # Assume 5% of 10Mbit bandwidth assignment to packets that was marked 1 in step 2. (10240*0.05 = 5120 = 512kbit for high bandwidth)
    - add_class 10:1 10:100 512kbit 51.2kbit 1 bounded
- # Assume 5% of 10Mbit bandwidth assignment to packets that was marked 2 in step 2 but such packets are allowed to bandwidth from other queue classes if available. (10240*0.05 = 5120 = 512kbit for high bandwidth)
    - add_class 10:1 10:200 512kbit 51.2kbit 2
- # Assume 70% of 10Mbit bandwidth assignment to packets that was marked 3 in step 2 but such packets are allowed to bandwidth from other queue classes if available. (10240*0.70 = 7168 = 7168kbit for high bandwidth)

- add_class 10:1 10:300 7168kbit 716.8kbit 3

3.2.3 Multi-Level Rate-Limiting

On the ingress side of the network interface card, traffic can be controlled using netfilters. Netfilters are a component of the Linux firewall system. This firewall subsystem provides packet filtering functionality that permits or denies packets from passing through based on security policies. The firewall rules are configured through the netfilter packet-filtering infrastructure that is a subsystem of the Linux kernel 2.4. The packet-filtering rules or firewall rules are defined by the /sbin/iptables commands. Netfilters examine a packet header's information including its source, destination addresses, ports, protocol type, any combination of the TCP flags, and MAC addresses. Iptables contain a function rate-limiting that can help to minimize the impact of DDoS attacks. For example, administrators can rate-limit a suspicious source to one packet per minute or control of the number of syn packets entering a network at any given time to ten SYN requests per second.

- /sbin/iptables -A INPUT -s 192.168.1.25 -m limit --limit 1/m –j ACCEPT
- /sbin/iptables -A INPUT –p TCP --syn -m limit --limit 10/s -j ACCEPT

If a source can be confidently identified as an attacker, it is more effective to drop all packets from that source. However, attack source identification is often difficult, especially with IP spoofing. Dropping all packets from a suspicious source may block a great deal of legitimate traffic. A flood mitigation mechanism that is able to stop most attack traffic, while having the smallest impact on legitimate traffic, is considered better than a mechanism that blocks a lot of legitimate traffic [For01]. To maximize the efficiency of rate-limiting, A2D2 proposes a multi-level rate-limiting mechanism.

It is conceivable that a network may generate a sudden burst of connection traffic. Such a burst lasts for a very short period of time while the connection is established. Traffic from initiating network hosts taper off over time as the abundance of traffic should flow from the servers serving files or streaming video to the clients. A multi-level rate limiting mechanism imposes stricter limits as the confidence that a source is malicious increases.

For example, if a source sends out 500 request packets per second, A2D2 can limit it to only 100 packets per second. The surge of requests is usually temporary. If the suspicious source continues to send out the maximum allowable rate, the firewall can further restrict the incoming packet rate to 50 packets per second. If the trend persists where a source continuously consumes the maximum allowable rate, the source will be blocked completely. This sample scenario can be achieved by the following steps:

1. Set up rate limiting levels by adding a new rule chain with the option –N and then adding the actual firewall rules under the chain using the –A option.

   - iptables –N level2

- iptables –A level 1 –match --limit 100/s -j ACCEPT
- iptables –N level1
- iptables –A level 1 –m --limit 50/s -j ACCEPT
- iptables –N level0
- iptables –A level0 –j DROP

2. Insert firewall rules against the source IP that is suspicious (12.23.34.45):

- iptables –I FORWARD –s 12.23.34.45 -j level2
- iptables – I FORWARD –s 12.23.34.45 –j level0

3.3 Autonomy System

To enable autonomous response, communication channels need to be established for the various components of the A2D2 detection and response systems. In addition, any firewall rules need to be autonomously applied and expired without administrators' intervention.

3.3.1 Rate-Limiting Configuration and Expiration

A rate-limiting configuration file is setup to enable administrators to define the basic parameters based on which rate-limiting can be automatically applied. The configuration file rateif.conf (rate limiting interface configuration) specifies six main elements:

1. The number of levels, the rate associated with each level, and the duration for which each rate level will be effective is shown in the following table:

Level	Rate (number of packets/sec) (Maximum level = 0) (Block level = -1)	Duration (days:hours:mins:secs)
L2	100	00:00:30:00
L1	50	00:00:30:00
L0	-1	01:12:00:00

Table 3.1 – Multi-Level Rate-Limiting Table

- Administrators are allowed to configure any number of levels he or she intends for the network. The minimum level is L0 and the administrator specifies the maximum level by putting a “0” in the rate column. A “-1” on the rate column indicates a block level where all packets associated with that level will be dropped by the firewall. In the above example, three levels are defined. Level 2 is the maximum level where packets will be limited to 100 packets per second. Once Level 2 rate is applied, the rate limiting will be in effect for 30 minutes before the rate-limiting firewall rule is deleted. During this 30 minutes, if the IDS indicates the packet source is still consuming all bandwidth allowable by Level 2, Level 1 rate limiting will be applied. Level 1 rate limiting will stay in effect for another 30 minutes. Should the IDS raises another alert, indicating the suspicious source continues to submit 50 packets per second, the final rate limiting level, Level 0, is applied and all packets from the source are blocked completely for 1 day and 12 hours. At the end of the period, the rate limiting rule will be deleted and packets will be allowed to access the network without limitations unless flooding is detected again.

2. The firewall server name or IP address
    - This option specifies the server on which the firewall is active and rate-limiting is applied.
    - The syntax is: SERVER firewall.server.edu
3. The port number at which the firewall server will listen for alerts from the IDS
    - The syntax is: PORT 6779
4. Log file name
    - The log file name where the firewall stores the changes made to iptables such as the insertion and the deletion of rate-limiting commands.
    - In case of accidental or intentional restarting of the firewall, the rate-limiting rules can be brought up-to-date by applying the commands in the logfile.

- The syntax is: SETTINGSFILE fwfile.dat

5. Location of iptables in the firewall system

   - The syntax is: /sbin/iptables

6. Time in seconds when the iptables rules are checked and updated

   - The syntax is: ALARMTIME 1
   - In this example, the iptables rules will be checked every second. If a particular rule has exceeded its effective duration, the rule will be deleted.

3.3.2 Snort Customization – FloodRateLimiter Preprocessor

To enable automatic rate-limiting, a FloodRateLimiter preprocessor add-on needs to be created for the flood preprocessor. After an attack source surpasses the initial flood threshold, an initial flood alert is sent to the firewall. The firewall applies rate-limiting as defined in the rateif.conf file against the attack source. The role of the FloodRateLimiter preprocessor is to keep track of the incoming packet rate of the presumed attack source after rate-limiting is applied. If the arrival rate of the attack source continues to reach the maximum allowable rate, another alert is sent to the firewall, which will apply a stricter rate-limiting level until the block level is reached. The setup of the FloodRateLimiter preprocessor is similar to that of the FloodIgnoreHosts preprocessor described in Section 3.1.5.

1. Add the following lines to the Snort plugbase.c file

```
void InitPreprocessor()
 {
        SetupFloodRateLimiter();
 }
```

2. Add the following lines to the snort.conf file

```
preprocessor flood-ratelimiter: <report time in sec> <rate-limiting level[n] threshold> <rate-limiting level[n-1] threshold> <rate-limiting level0 threshold>
# This configuration option allows administrators to specify the rate-limiting threshold after the initial flood alert is raised. In the following example, the IDS will check for the rate for 10 seconds, and see if the first rate-limiting threshold of 100 packets has arrived within these 10 seconds. Another alert is raised and sent to the firewall if the condition is met. Then the IDS will check if the server receives 50 packets in the 10 seconds after the 2nd alert is raised and the stricter rate-limiting level is applied.
#preprocessor flood-ratelimiter: 10 100 50
#The configuration of these numbers should correspond with the rate-limiting levels set on the rateif.conf file
```

3. In spp_flood.h, add

```
void SetupFloodRateLimiter(void)
void FloodRateLimiterInit(u_char *)
# The FloodRateLimiterInit function creates the preprocessor data structure
```

4. In spp_flood.c, register the preprocessors by adding the following function:

```
void SetupFloodRateLimiter(void)
{
        RegisterPreprocessor("flood-ratelimiter",
FloodRateLimiterInit);
}
```

A separate data structure is created to keep track of the connections that have passed the initial flood threshold as described in

Figure 3.7. The FloodRateLimiter preprocessor follows similar logic as the Flood Preprocessor described in Section 3.1.4.5 on page 40. The FloodRateLimiter preprocessor is called by the FloodPreprocFunction after the initial flood threshold is passed and a flood alert is raised. The FloodRateLimiter preprocessor then takes over surveillance of the flood connection until the flooding is ended. Some key functions related to the FloodRateLimiter logic are also similar to those of the flood preprocessor explained on page 40. These functions are:

- RateList *CreateRateList(void)
- void ExpireRateConnections(RateList *ratelist, struct spp_timeval watchPeriod, struct spp_timeval currentTime)

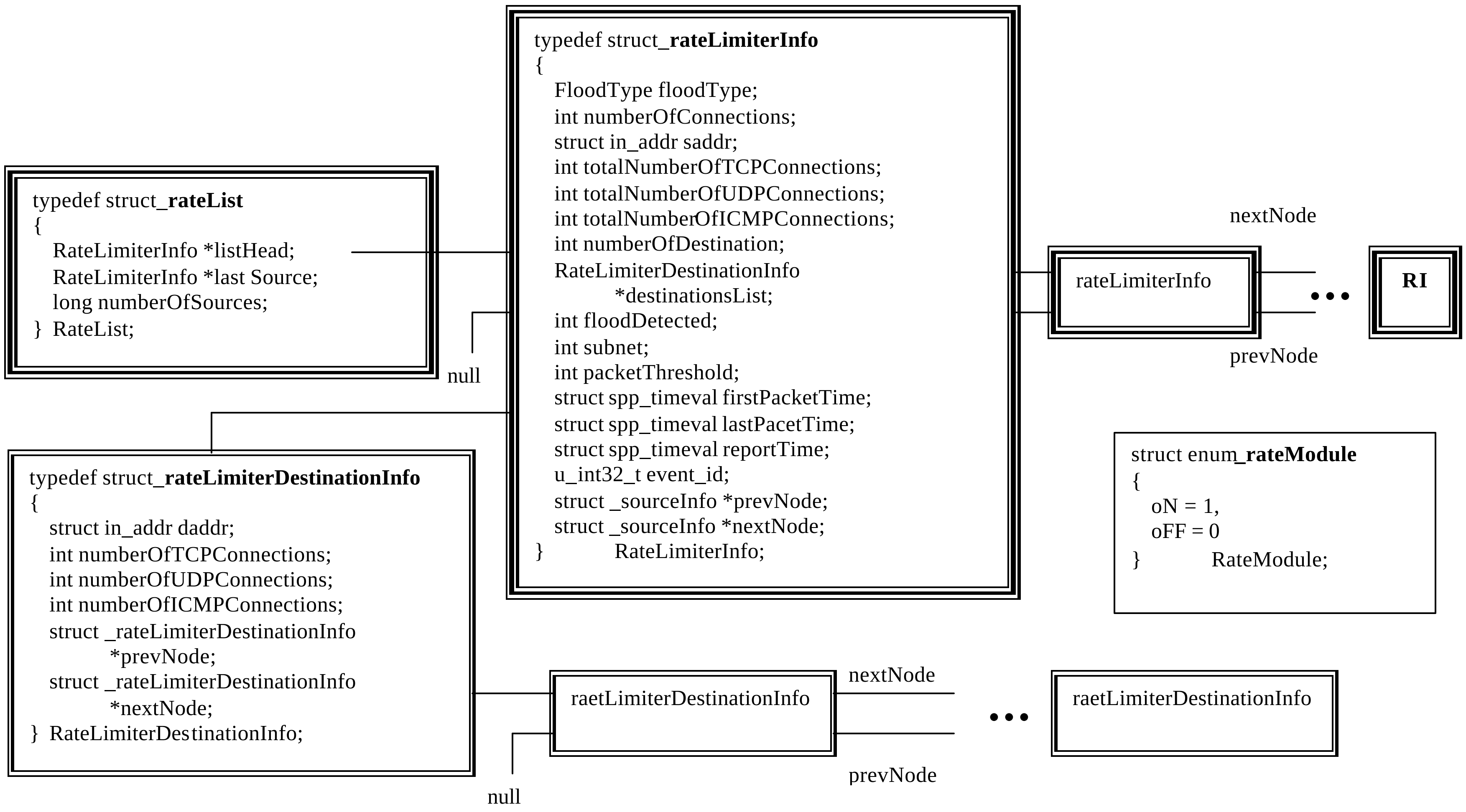
- int updateRateLimiterConnection(RateList *rateList, Packe *p, FloodType floodType)
- void ClearRateConnectionInfoFromFloodSource(RateLimiterSourceInfo *currentSource);

3.3.3 Alert Interface

Snort has a real-time alerting capability as well, incorporating alerting mechanisms for syslog, a user specified file, a UNIX socket, or WinPopup messages to Windows clients using Samba's smbclient program. For A2D2, Snort is directed to send alert messages to a UNIX socket using the command ./snort –A UNSOCK.

An interface program, report.c is written to accept the Snort’s alert messages on the Snort hosts and to send the messages to the specific port of the firewall machine. The report.c program also logs the messages to a log file specified by users for the administrators’ evaluation.

Another program, rateif.pl, resides in the firewall server and listens to messages sent by the Snort host to the firewall server’s designated port. Once the message is received, the alert message is parsed for the source IP address of the attack host or subnet. Multi-level rate-limiting is then applied to the IP address based on the rateif.conf configuration file.



```
typedef struct_rateList
{
   RateLimiterInfo *listHead;
   RateLimiterInfo *last Source;
   long numberOfSources;
}  RateList;
```

```
typedef struct_rateLimiterInfo
{
   FloodType floodType;
   int numberOfConnections;
   struct in_addr saddr;
   int totalNumberOfTCPConnections;
   int totalNumberOfUDPConnections;
   int totalNumberOfICMPConnections;
   int numberOfDestination;
   RateLimiterDestinationInfo
            *destinationsList;
   int floodDetected;
   int subnet;
   int packetThreshold;
   struct spp_timeval firstPacketTime;
   struct spp_timeval lastPacetTime;
   struct spp_timeval reportTime;
   u_int32_t event_id;
   struct _sourceInfo *prevNode;
   struct _sourceInfo *nextNode;
}           RateLimiterInfo;
```

```
typedef struct_rateLimiterDestinationInfo
{
   struct in_addr daddr;
   int numberOfTCPConnections;
   int numberOfUDPConnections;
   int numberOfICMPConnections;
   struct _rateLimiterDestinationInfo
            *prevNode;
   struct _rateLimiterDestinationInfo
            *nextNode;
}  RateLimiterDestinationInfo;
```

```
struct enum_rateModule
{
   oN = 1,
   oFF = 0
}           RateModule;
```

Figure 3.7 – FloodRateLimiter Preprocessor Key Data Structure

Chapter 4

A2D2 Implementation

A test-bed is setup based on the A2D2 design presented in Chapter 3. As presented in Figure 4.1 on page 67, the A2D2 implementation test-bed is divided into three zones:

- The attack network
- The autonomous defense network (A2D2)
- The client network

4.1 The Attack Network

The attack network is made up of five Red Hat Linux machines. The DDoS tool chosen in the test-bed is Stacheldraht version 4.0. One computer in the attack network (Saturn) serves the dual-role of the "Master Client" and the handler. The other four computers are the agents. These four computers are connected to the A2D2

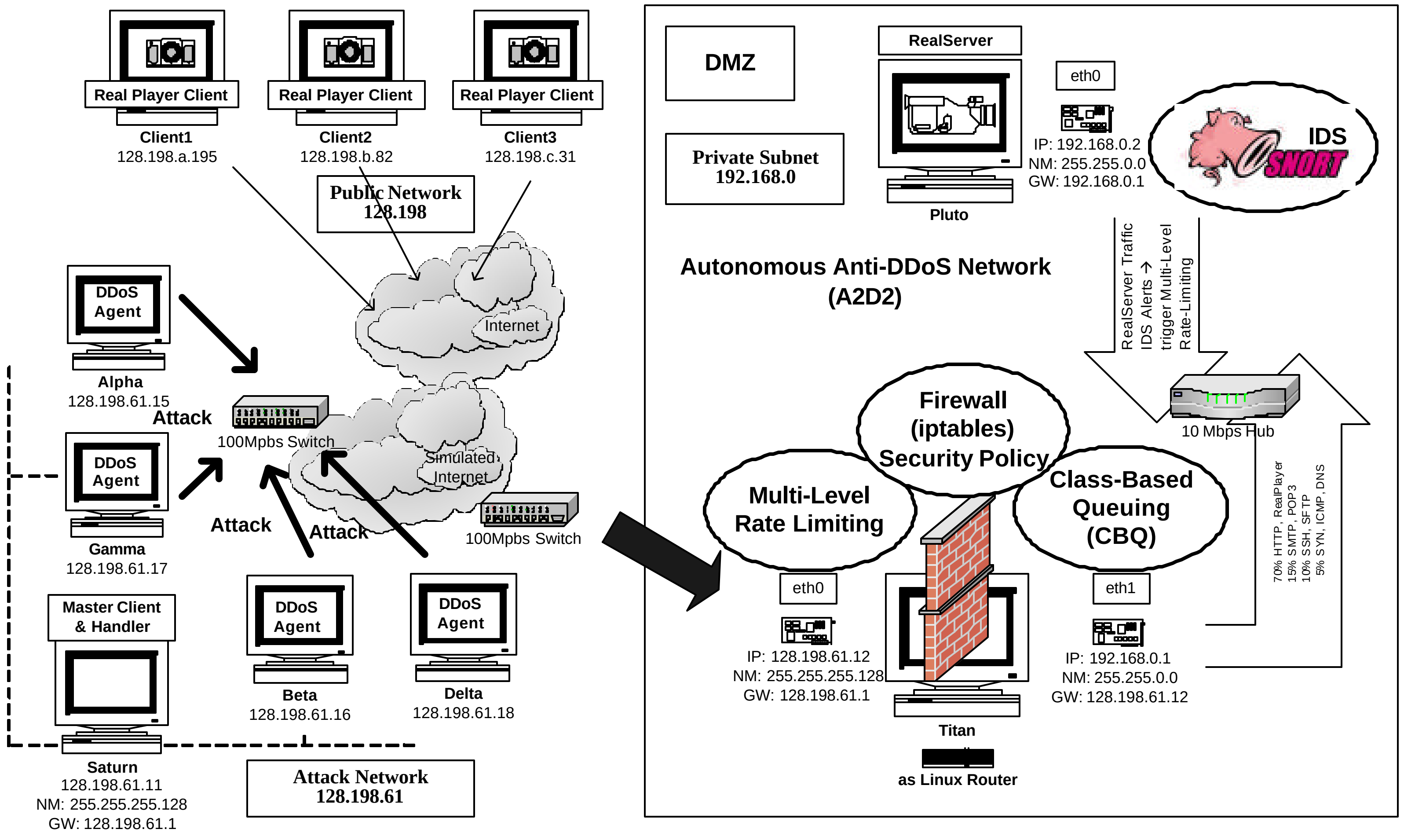


Figure 4.1 – A2D2 Implementation Test-bed

DMZ through two 100 Mpbs switches that emulate the role of the Internet. Such a setup allows us to isolate the test-bed traffic from the actual Internet and ensure no attack traffic can leak into the Internet during preliminarily testing.

4.1.1 Attack Network Systems Specifications

The computer “Saturn” is used for attack coordination. Saturn assumes the dual roles of the “Master client” and “Handler”. The system specifications for Saturn are:

- Model: HP Vectra VL
- CPU: Intel Pentium III 501.143 MHz
- RAM: 255 MB
- Hard Drive: 8455 MB
- Network Interface: 3com 100 Mb
- OS: Red Hat Linux 7.3 2.96-110
- Linux Kernel Version: 2.4.18-10

Four attack agents are used as described in Figure 4.1. The system specifications for Alpha, Beta, Gamma and Delta are:

- Model: HP Kayark XA
- CPU: Intel Pentium II 233.344 MHz
- RAM: 93 MB

- Hard Drive: 2564 MB
- Network Interface: 3com 100 Mb
- OS: Red Hat Linux 7.1 2.96-98 for Alpha and 7.1 2.96-85 for the other 3 agents
- Linux Kernel Version: 2.4.9-21 for Alpha and 2.4.3-12 for the other 3 agents

4.1.2 Attack Tool – StacheldrahtV4

The attack tool chosen for the A2D2 test-bed is Stacheldraht version 4.0. Common DDoS attack tools such as Trinoo, Trible Flood Network (TFN), Trible Flood Network 2000 (TFN2K), mstream and Stacheldraht are similar to one another [Dit99]. Among DDoS tools, StacheldrahtV4 is considered stable and sophisticated and is able to launch attacks in ICMP, UDP and TCP protocols. Stacheldraht can be downloaded freely from the Internet [ASTA00]. Most DDoS tools obtained from the Internet have been slightly altered to justify the posting as “for education purpose” instead of for attack distribution. These alterations in the attack tools source code generally involve the following aspects:

- Communication ports among master client servers, handlers and agents
- IP addresses of the master client servers and handlers
- Login passwords and encryption
- Payload contents between the master client servers and the handlers

- Payload contents between the handlers and agents

4.1.3 A2D2 Implementation of StacheldrahtV4

In the A2D2 test-bed attack network, all Stacheldraht source files are extracted under the main folder “Stacheldrahtv4” in Saturn. The handler and agent source files reside directly under the “Stacheldrahtv4” main directory. Two additional folders “leaf” and “telnetc” are created under the main directory. The folder “leaf” contains the source files for agents while the master server client files are stored under “telnetc”. The main files for master server client, handler and agents are “client.c”, “mserv.c” and “td.c” respectively. These file names constitute the executable file names after compilation. To make the system distributed, the “td” executable is saved in all DDoS agents in the test-bed, namely host “Alpha”, “Beta”, “Gamma” and “Delta”. The agent process is then started with the command “./td”. Since Saturn serves as both the master client server and the handler, the handler process has to be started in Saturn with “./mserv”. Then, the DDoS attack can be launched by the client. In the “Stacheldrahtv4/telnec” directory in Saturn, the command “./client <handler servername>” is executed. After successful login, the “client” program can command the handler to coordinate various attacks through the agents.

4.1.4 Other Possible Attack Tools

The attack network setup can easily support DDoS attacks using other DDoS and DoS tools. For example, Trinoo and TFN2K have identical architecture as

StaheldrahtV4 and their client and handler programs can be installed in Saturn while their agent executables can be installed in the four agent computers: Alpha, Beta, Gamma, and Delta. A large variety of basic DoS attack tools can be easily downloaded from the Internet [ASTA00]. Some examples of DoS tools are ipsyn.c, udpf.c, pepsi.c (UDP flooder) and synack.c. Each of them launches a type of DoS attack as indicated by its file name. As mentioned in Section 1.2 on page 4, DDoS attacks are simply DoS attacks being launched from a number of clients instead of from one host. Within the A2D2 test-bed, these DoS attacks can be made "distributed" by installing the executable of such DoS tools as ipsyn and udpf in the four "agent" computers and starting the DoS attack from each agent.

4.2 The Autonomous Defense Network (A2D2)

A simple DMZ is set up with two Linux computers to represent the A2D2 design described in Chapter 3. The gateway computer "Titan" is configured as a Linux router and serves as the firewall at the entry of the DMZ. Packets that successfully pass through the firewall enter the private DMZ subnet 192.168.0. Inside the private subnet, one server "Pluto" is installed with the RealServer application and the web server "Apache". The RealServer connects to Titan through a 10 Mbps hub inside the private network. Ideally, the IDS can be placed in a separate host in the subnet and monitor internal traffic in a promiscuous mode. For simplicity, the open source IDS Snort is also configured in Pluto in this test-bed.

4.2.1 Defense Network Systems Specifications

The firewall gateway (Titan) and the RealServer/IDS (Pluto) use identical computer models:

- Model: HP Vectra VL
- CPU: Intel Pentium III 501.143 MHz
- RAM: 255 MB
- Hard Drive: 8455 MB
- Network Interface: 3com 100 Mb
- OS: Red Hat Linux 7.3 2.96-110
- Linux Kernel Version: 2.4.18-10
- Network Monitoring Tool: IPTraf Version 2.5.0
- (Pluto) IDS: Snort 1.8.6
- (Pluto) Media Streaming System: RealServer 8
- (Pluto) WebServer: Apache-1.3.23

4.2.2 Defense Network Gateway Policy and CBQ Setup

The firewall (Titan) practices the deny security policy and only routes TCP and UDP external traffic that is targeted to specific service ports on Pluto. Allowable protocol and destination ports are [IANA02]:

- TCP/UDP-port 21, 22 and 23 for FTP, SSH, and Telnet services
- TCP/UDP-port 25 for SMTP services
- TCP/UDP- port 42 for DNS services
- TCP/UDP-port 80, 8080 for HTTP and RealPlayer traffic
- Other ports opened for remote administration of RealServer, Snort and other applications.
- For testing purpose, iptables are also configured to allow ICMP packets.

Class-Based Queuing is implemented on Titan to manage outbound traffic into the private network. Seventy percent of the internal link bandwidth is allocated to HTTP and RealPlayer traffic, SMTP is allowed 15% of the bandwidth, SSH, Telnet and FTP have 10% of the bandwidth and SYN and ICMP traffic is bounded to 5% of the network link bandwidth. In addition, the rateif.pl program in Titan opens the port 6779 to listen for alerts from the IDS.

4.2.3 Defense Network IDS and Rate-Limiting Setup

The flood preprocessor, flood-ignorhosts preprocessor, and the flood-ratelimiter preprocessor are compiled with Snort in Pluto. When Snort identifies any intrusion such as a flood attack or a portscan attempt, Snort sends an alert to port 6779 of Pluto. Upon receipt of the alert, Titan will implement the multi-level rate limiting. The rate.conf file in Titan specifies five levels:

Level	Rate (number of packets/sec) (Maximum level = 0) (Block level = -1)	Duration (days:hours:mins:secs)
L4	0	02:00:00:00
L3	100	00:00:30:00
L2	50	00:01:00:00
L1	-1	00:06:30:00
L0	-1	01:12:00:00

Table 4.1 – Titan Multi-Level Rate-Limiting Table

Five levels of rate-limiting are defined with the "maximum" level being Level 4. The "maximum" level is technically not a "rate-limiting" level, but a log level. IP addresses that have reached the strictest level (L0) and have subsequently been cleared as an attack source will be logged as L4 for the specified period of time (For example, two days according to Table 4.1). Source assigned with L4 have full access through the firewall and no rate limiting is applied to them. However, if the IDS raises an alert related to a L4 source address within the 2-day period, the level of this source address will be immediately dropped to L0. This source address will then be blocked for 1 day and 12 hours without going through L3 to L1.

When the IDS raises the first alert regarding an attack source, the source will be assigned with the level L3 and be limited to a rate of 100 packets per second. The IDS will check if the attack source is still sending packets at the fullest rate as defined by L3. If the L3 rate is used to its fullest extent by the source, the IDS will raise another alert that drops the attack source to L2. However, if the number of packets sent by the attack source is lower than the maximum allowable L3 rate, the rate limiting will expire for the source in 30 minutes. Similar checks and mitigation processes are applied for each rate-limiting level. In L1, the source is blocked for 6 hours and 30 minutes. After the blocking expires, another alert for this source will drop the attack source to L0 that will block the attack source for a longer period of time than L1.

The multi-level rate limiting technique can ensure the highest level of service to legitimate traffic during a possible attack. A source identified as malicious is given plenty of opportunities to prove its innocence. A mitigation system that is able to stop most attack traffic while having the smallest impact on legitimate traffic is considered a better design than devices that block a lot of attack traffic along with legitimate traffic [For01]. In fact, Forristal [For01] describes one flood mitigation method called the floodgate approach. In the floodgate method, the firewall blocks all traffic from potential attackers but occasionally lets small amounts of traffic pass through for a short period of time. The assumption is that the small amount of traffic allowed through is likely to consist of both legitimate and attack traffic. The goal is to keep the target system from being overloaded while accommodating some legitimate traffic. The multi-level rate-limiting design implemented in the A2D2 test-bed provides a sophisticated means to ensure maximum QoS experienced by legitimate clients even in situations where false alerts are raised.

4.3 The client network

The client network of the A2D2 test-bed consists of three RealPlayer clients from three different networks. These clients access the RealServer (Pluto), which resides in the A2D2 DMZ. To show QoS experienced by the A2D2 clients, testing tools are installed in the client computers to measure the packets received by the RealPlayer clients during a multimedia session. During a DDoS attack, a successful autonomous mitigation carried out by A2D2 will enable the clients to enjoy a continuous, uninterrupted packet flow serviced by the RealServer.

4.3.1 Real Clients Specifications

Any computer can be used as a Real Client to the A2D2 network. In this test-bed, two Linux hosts and one Windows 2000 system are used as clients. All clients use RealPlayer 8 as their media player. All Real Clients connect to the RealServer Pluto to access the video clip “simpsons.rm”. The video clip is located in the “Content” folder under the main “Realsystem” directory and is played by the clients for about 10 minutes

4.3.2 Client Bandwidth Measurement Tools

A bandwidth measurement program “plot.pl” is written and installed in the Linux clients to capture bandwidth usage information. The “plot.pl” program extracts information from the “/proc/net/dev” file of the Linux system. The “/proc/net/dev” file provides real-time interface activities such as the number of packets and the amount of bytes received and transmitted by the network interface. The information is parsed and displayed in a graphical format using the drawing tool gnuplot version 3.7.1

[Gnuplot]. In addition to the plot.pl and gnuplot programs, IPTraf Version 2.5.0 is also used in Linux clients to monitor network activities.

Chapter 5

Performance Results and Analysis

5.1 Test Scenarios

Eight groups of test scenarios are conducted with the A2D2 test-bed and results are compiled. For each scenario, a minimum of three test runs is executed to verify performance consistency. Since the results are relatively consistent, only the last set of performance results are presented in this section. The eight test scenarios are:

1. Baseline

    - Data is collected for "normal" A2D2 network activities where three RealClients are accessing the RealServer. The network is not experiencing any DDoS attack and no mitigation strategy is applied. This data is considered to be the A2D2 baseline traffic.

2. Short 1-minute attack with no mitigation strategy

    - DDoS attacks are launched for one minute to show the effect of the attack on the QoS of the RealClient. The attack protocols used are UDP, TCP and ICMP.

3. Non-stop attack with no mitigation strategy

    - DDoS attacks are launched for the entire duration of the test case to show the effect of a non-stop attack on the QoS of the RealClient. The attack protocols used are UDP, TCP and ICMP.

4. Non-stop UDP attack with security policy

    - The security policy of A2D2 is applied for this test scenario. Only traffic that targets the active UDP and TCP service ports at the RealServer is allowed to pass through the firewall. The RealClient's QoS is evaluated during a non-stop UDP attack against the A2D2 network. The security policy is expected to stop most attack tools that generate random destination ports for their attack traffic.

5. Non-stop ICMP attack with security policy

    - Since A2D2 does not implement any policy to block ICMP packets, a test run is conducted to show the effect of a non-stop ICMP packet even when the firewall security policy script is applied. The hypothesis is that the QoS experienced by RealClient will suffer.

6. Non-stop ICMP attack with security policy and CBQ
    - The goal of this test scenario is to show the effectiveness of CBQ against attacks that cannot be countered by basic firewall policy.
7. Non-stop TCP-SYN attack with security policy and CBQ
    - This test scenario verifies that the A2D2 security policy and CBQ can also mitigate attacks based on protocols other than UDP and ICMP.
8. Non-stop TCP-SYN attack with security policy, CBQ, and autonomous multi-level rate-limiting
    - This final test scenario demonstrates the intrusion tolerance of the A2D2 network and how all mitigation techniques collaborate with one another within the A2D2 network.

5.2 Test Data Collection

Performance data are collected from a Linux A2D2 client, the A2D2 gateway firewall and the A2D2 RealServer. The tools used for data collection at these hosts are IPTraf. At the Linux A2D2 client, the “plot.pl” program mentioned in Section 4.3.2 is executed and the gnuplot utility is used to graph the network activity. If unusual traffic is observed, “ethereal” is employed to identify the source and activity of the traffic pattern that deviates from baseline and cannot be accounted for by the test scenarios.

The information collected at the Linux A2D2 client, A2D2 gateway firewall and the A2D2 RealServer is:

- Peak total activity in number of packets per second
- Peak incoming rate in number of packets per second
- Peak outgoing rate in number of packets per second
- Average rate total in number of packets per second
- Average Incoming rate in number of packets per second
- Average Outgoing rate in number of packets per second

In addition to the above parameters, the following information is registered at the Linux A2D2 RealClient:

- The number of packets received per second at the A2D2 client including RealPlayer traffic and non-RealPlayer traffic
- RealPlayer Statistics: The total number of packets received by the RealPlayer when playing the simpsons.rm video clip for about 10 minutes.
- RealPlayer Statistics: The total number of packets recovered (received after retransmission requests were sent) by the RealPlayer for the duration of each test run.
- RealPlayer Statistics: The total number of packets lost by the RealPlayer for the duration of each test run.

- RealPlayer Statistics: The total number of retransmission requests made by the RealPlayer during each test scenario.
- RealPlayer Statistics: The total number of retransmissions received by the RealPlayer during each test scenario.
- RealPlayer connection time-out conditions and screen quality.

5.3 Test Results

The test results collected by IPTraf and RealPlayer Statistics for each test scenario are presented in Table 5.1. As mentioned in Section 5.1, the 1-minute attack and the non-stop DDoS attack scenarios were launched using TCP, UDP, and ICMP packets. The effect of the attack is similar regardless the protocol used. Therefore, data collected from the UDP attack test runs is presented in this report as representative results for the 1-minute and non-stop attack scenarios.

The data collected at the A2D2 Firewall Gateway computer (Titan) is provided in Table 5.1 but should be considered as unreliable. IPTraf logging process is resource-intensive. Two IPTraf logging processes are started, one for eth0, the network interface card (NIC) connected to the Internet interface, and one for eth1, the NIC connected to the internal DMZ network. In addition to IPTraf logging, the firewall gateway (Titan) is also responsible for packet filtering, policy enforcement, network address translation, packet classifying, scheduling, and forwarding, as well as performing multi-level rate limiting. The gateway computer is also the target of the

DDoS attacks. During testing, the firewall gateway often “hung” and became non-responsive to mouse and keyboard inputs. However, the firewall functions, including that of rate limiting, firewall policy enforcement, and CBQ are carried out accurately, as indicated by the packets routed to the A2D2 RealServer (Pluto) and the A2D2 clients. Therefore, the data presented by the IPTraf logs for eth0 and eth1 of the firewall should be viewed with caution. Instead, the analysis of the performance results should mainly rely on information collected at the A2D2 RealServer (Pluto) and the A2D2 RealPlayer clients.

Network Environment								
	Baseline	**Attack Scenario**						
Attack Protocol	N/A	UDP	UDP	UDP	ICMP	ICMP	TCP-SYN	TCP-SYN
Attack Duration	N/A	1 minute	Non-stop	Non-stop	Non-stop	Non-stop	Non-stop	Non-stop
A2D2 Mitigation								
Techniques	N/A	N/A	N/A	Policy	Policy	Policy CBQ	Policy CBQ	Policy CBQ Rate
A2D2 Client RealPlayer Statistics								
Received (packets)	23445	17869	8039	23407	7127	23438	22179	23444
Recovered (packets)	0	121	35	0	4	0	2641	40
Lost (packets)	0	1808	2557	0	2101	0	1449	9
Retransmission Requests	0	1929	2592	0	2105	0	4090	49
Retransmission Received	0	121	35	0	4	0	2641	40
Connection Time-out	N/A	Screens frozen but recovered	All clients timed-out around 100 seconds into the attack	N/A	All clients timed-out around 100 seconds into the attack	N/A	No frozen screen or timed-out. Noticeable inferior screen quality	N/A

Table 5.1 – A2D2 Test-bed Performance Results

(Table Continues on Next Page)

Network Environment								
	Baseline	Attack Scenario						
Attack Protocol	N/A	UDP	UDP	UDP	ICMP	ICMP	TCP-SYN	TCP-SYN
Attack Duration	N/A	1 minute	Non-stop	Non-stop	Non-stop	Non-stop	Non-stop	Non-stop
A2D2 Mitigation								
Techniques	N/A	N/A	N/A	Policy	Policy	Policy CBQ	Policy CBQ	Policy CBQ Rate
A2D2 Client IPTraf Statistics (packets/second)								
Peak Total Activity	103.40	376.40	300.80	111.6	216.60	104.00	103.20	102.8
Peak Incoming Rate	102.40	102.00	102.00	106.40	102.20	102.00	102.20	101.40
Peak Outgoing Rate	5.40	295.40	239.80	5.40	181.00	5.40	15.60	7.20
Average Rate Total	40.87	42.02	21.51	41.68	21.07	41.36	45.47	41.30
Avg Incoming Rate	39.40	34.96	17.37	40.23	14.96	39.95	37.49	39.76
Avg Outgoing Rate	1.47	7.06	4.14	1.45	6.11	1.42	7.98	1.54
A2D2 RealServer/IDS IPTraf Statistics (packets/second)								
Peak Total Activity	295.80	1241.20	1238.80	297.00	1125.40	296.00	1132.80	1096.40
Peak Incoming Rate	6.40	1059.40	1129.60	50.80	1124.40	100.60	503.20	495.80
Peak Outgoing Rate	292.80	348.801	446.40	294.00	343.80	293.00	634.60	600.60
Average Rate Total	113.04	185.13	788.74	170.45	575.39	211.53	705.21	129.66
Avg Incoming Rate	3.13	98.80	715.92	31.19	505.25	53.87	299.34	10.87
Avg Outgoing Rate	109.91	86.33	72.81	139.25	70.14	157.66	405.87	118.79

Table 5.1 Continued – A2D2 Test-bed Performance Results

(Table Continues on Next Page)

Network Environment								
	Baseline	**Attack Scenario**						
Attack Protocol	N/A	UDP	UDP	UDP	ICMP	ICMP	TCP-SYN	TCP-SYN
Attack Duration	N/A	1 minute	Non-stop	Non-stop	Non-stop	Non-stop	Non-stop	Non-stop
A2D2 Mitigation								
Techniques	N/A	N/A	N/A	Policy	Policy	Policy CBQ	Policy CBQ	Policy CBQ Rate
A2D2 Firewall/Gateway eth0 IPTraf Statistics (packets/second)								
Peak Total Activity	298.20	305.60	575.80	1435.20	6629.40	5980.60	199.60	5344.20
Peak Incoming Rate	7.00	303.60	574.60	1410.40	6611.60	5812.80	25.20	5339.00
Peak Outgoing Rate	293.80	294.80	294.40	282.80	294.40	295.20	197.00	294.20
Average Rate Total	116.33	84.08	95.37	835.61	2927.18	3603.20	21.99	2735.57
Avg Incoming Rate	4.00	16.07	63.93	787.46	2883.08	3482.80	1.08	2702.47
Avg Outgoing Rate	112.34	67.38	31.44	48.15	44.10	120.4	20.91	33.09
A2D2 Firewall/Gateway eth1 IPTraf Statistics (packets/second)								
Peak Total Activity	297.80	296.20	295.40	297.20	1123.60	297.00	199.00	290.60
Peak Incoming Rate	294.40	293.20	292.40	294.00	294.20	293.60	197.00	287.00
Peak Outgoing Rate	7.40	29.20	26.00	7.40	1123.60	88.00	6.60	11.40
Average Rate Total	115.17	70.54	39.32	41.38	440.99	143.78	18.46	43.75
Avg Incoming Rate	111.98	67.07	30.84	39.77	42.45	109.03	17.93	42.30
Avg Outgoing Rate	3.19	3.47	8.48	1.61	398.54	34.74	0.53	1.44

Table 5.1 Continued – A2D2 Test-bed Performance Results

5.4 Results Analysis

The test results presented in Section 5.3 show that the RealPlayer application of the A2D2 Linux client should receive around 23,000 packets from the RealServer during the 10-minute video clip. The exact number of packets received can ranged from 23,445 to 23,438 for scenarios where complete QoS was ensured and no packets were lost. This is due to the fact the video clips were stopped at around 10 minute playing time. The exact stopped time may deviate slightly in seconds from scenario to scenario. In addition to the total packet received, a complete transmission is indicated by the number of packet recovered, number of packet lost, the number of retransmission requests and the number of retransmission received (recovered). The peak number of incoming packets received by the A2D2 client should be just over 100 packets per second while the average incoming rate was around 39 to 40 packets per second. The normal traffic pattern experienced by the A2D2 clients is graphically presented in Figure 5.1.

Figure 5.2 shows the histogram of traffic at A2D2 clients during the 1-minute attack test scenario. The initial traffic pattern of the A2D2 clients resembled the baseline pattern presented in Figure 5.1. However, packet reception was completely interrupted during the 1-minute attack. Minimum packets were received and Figure 5.2 showed a slight burst of traffic around 180 seconds attempting to refill the RealPlayer buffer.

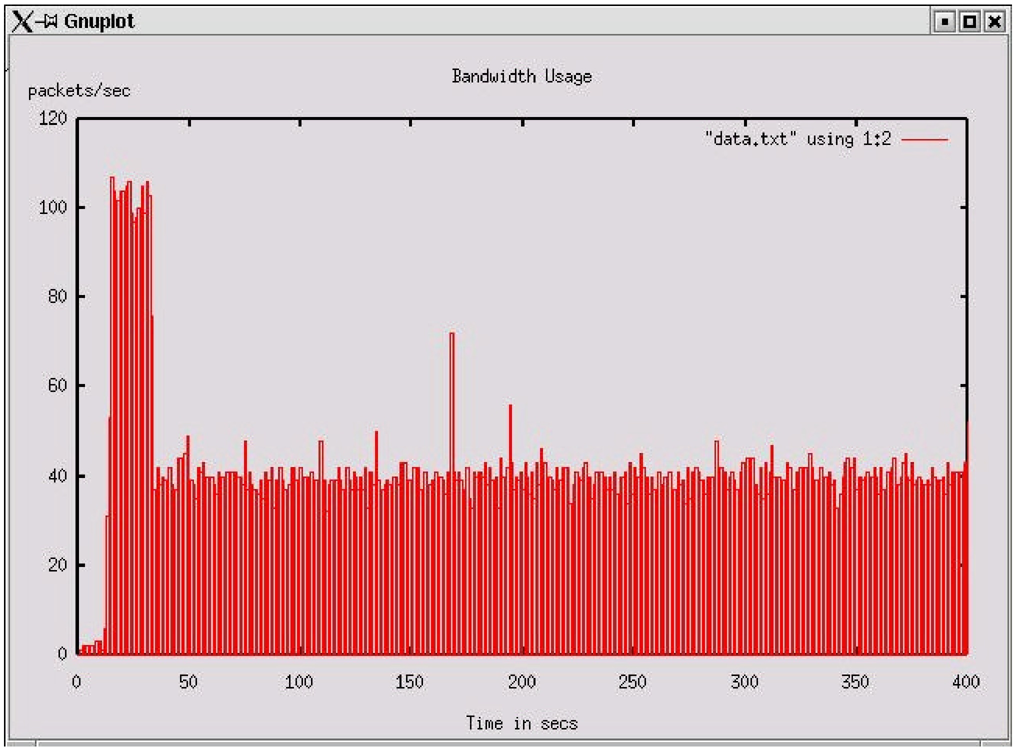


Figure 5.1 – QoS Experienced by A2D2 Client During Experiment Normal Activity

IPTraf results presented in Table 5.1 indicated that a large number of packets were being sent out from A2D2 clients with the peak outgoing rate reaching close to 300 packets per second. The considerable increase in outgoing packets represented the retransmission requests generated by RealPlayer that totaled 1,929 requests during the 1-minute attack. Unfortunately, only 121 packets were recovered out of the 1,929 requests. The video quality was significantly impacted and frozen screen shots were observed. However, as the attack stopped around 230 seconds into the experiment, packet reception resumed and the quality of screen shots recovered.

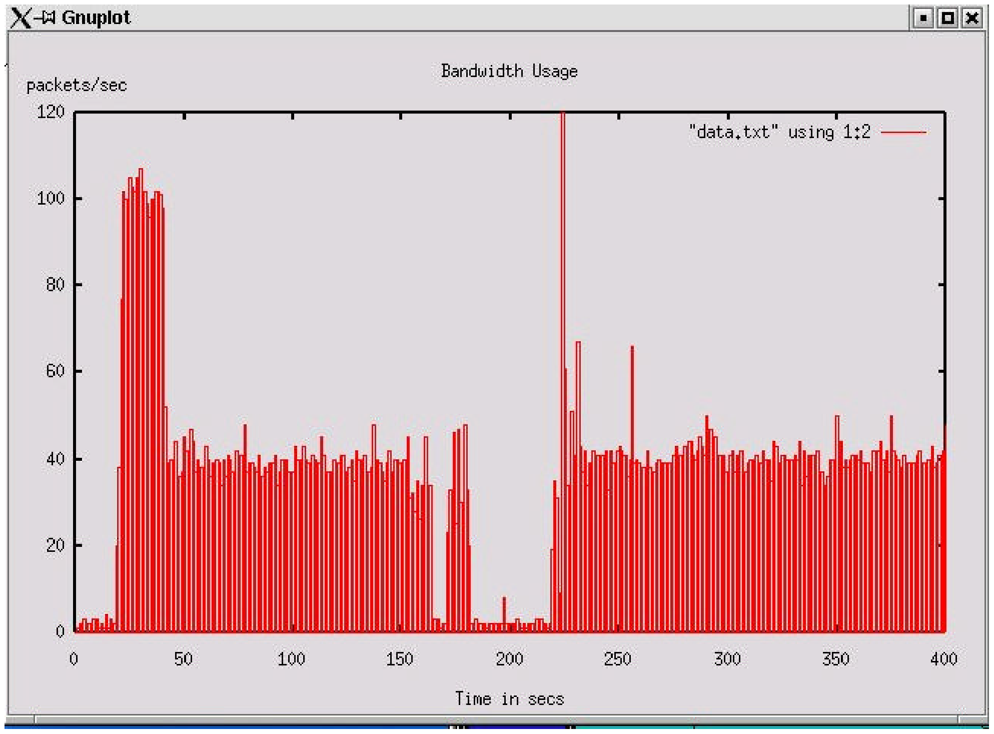


Figure 5.2 – QoS Experienced by A2D2 Client During 1-Minute DDoS Attack

During a non-stop DDoS attack, all RealPlayer clients were timed out due to the substantial loss of packets. A total of about 8000 packets were received by the RealPlayer as opposed to the 23,000 packets received during normal network activity. Only 35 packets were recovered out of 2,592 retransmission requests. The average incoming rate of the RealPlayer dropped to 17.37 packets compared to the average rate of 39 to 40 packets received during normal operations. The severely degraded QoS experienced by the RealPlayer client can be observed from Figure 5.3.

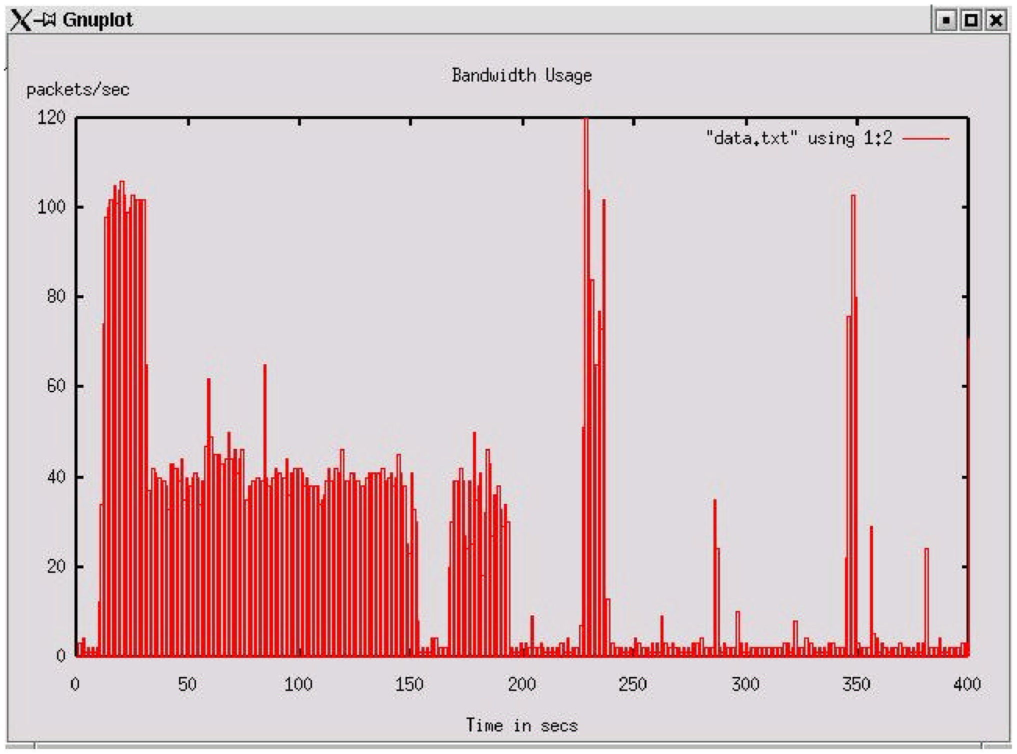


Figure 5.3 – QoS Experienced by A2D2 Client During Non-Stop DDoS Attack

The effectiveness of the A2D2 security policy is evidenced by the fourth test scenario where an UDP DDoS attack was launched against the A2D2 with a tight UDP policy enabled. Only port 22, 8080, 7070 and a few administrative ports were opened for UDP traffic. The base firewall policy was set as “DENY”. Since Stacheldraht generated UDP packets with random destination ports, any UDP attack packets that did not target the valid ports were discarded by the firewall and never reached the RealServer. As shown in Figure 5.4, the A2D2 client enjoys QoS similar to that of the baseline scenario. In fact, the RealPlayer received around 23,000 packets in total for the 10-minute video clip, sent out no retransmission requests and experienced no packet loss.

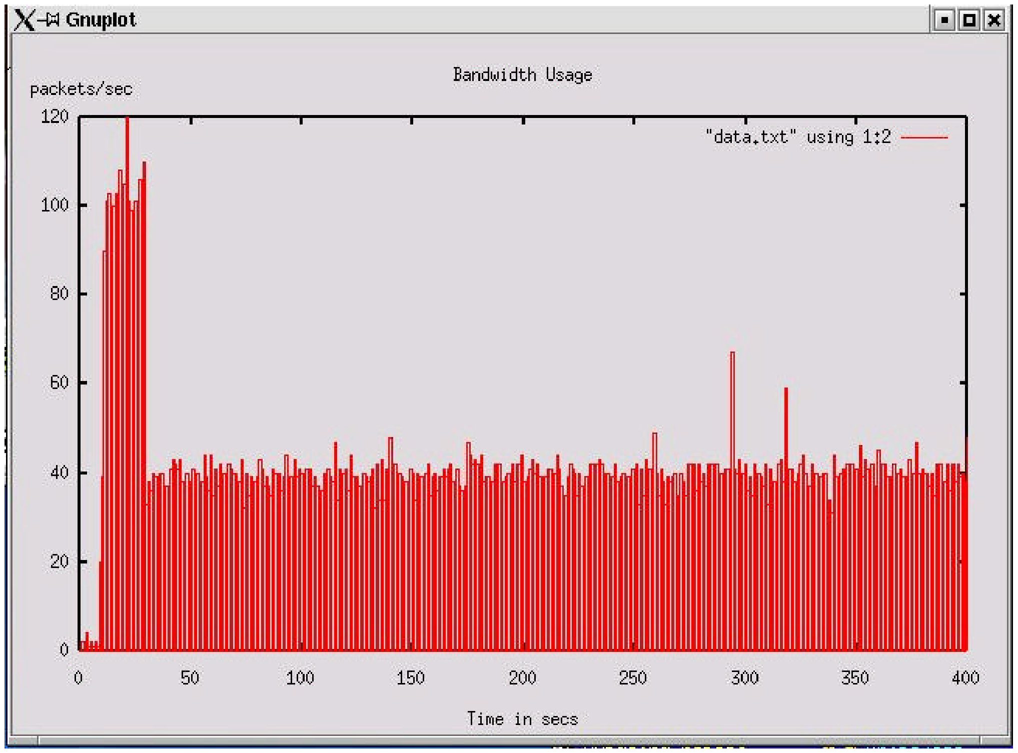


Figure 5.4 – QoS Experienced by A2D2 Client During Non-Stop UDP DDoS Attack with A2D2 Firewall Policy Enabled

Although firewall policy is effective, administrators may not be able to foresee all possible attacks and set up policies accordingly. In an event where the security policy is incomplete or where attack traffic targets the open service ports, the firewall gateway will still forward the attack traffic onto the DMZ. The fifth test scenario launched an ICMP DDoS attack against the A2D2. Since the A2D2 did not have a specific policy governing ICMP packets, the ICMP DDoS attack interrupted the A2D2 QoS entirely as shown in Figure 5.5. In fact, the effect is similar to that of test scenario 3 where a non-stop DDoS attack was launched against a network with no mitigation technique. During the ICMP Stacheldraht attack, a total of 7127 packets are received by the RealPlayer instead of the 23,000 packets normally transmitted for the entire 10-minute video clip. Only four packets were recovered out of the 2,105 retransmission requests sent.

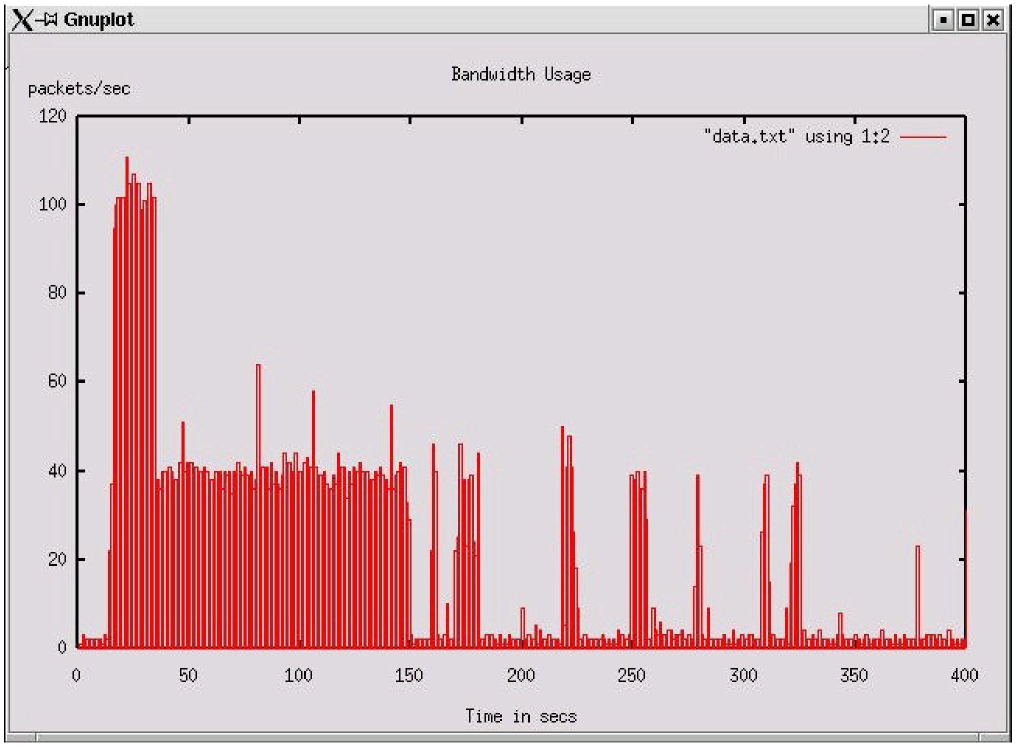


Figure 5.5 – QoS Experienced by A2D2 Client During Non-Stop ICMP DDoS Attack with A2D2 Firewall Policy Enabled

By enabling CBQ, the QoS experienced by A2D2 RealClient was returned to that of the baseline condition as showed in Figure 5.6. The A2D2 RealClient was able to receive all 23,000 packets related to the video clip without retransmission requests.

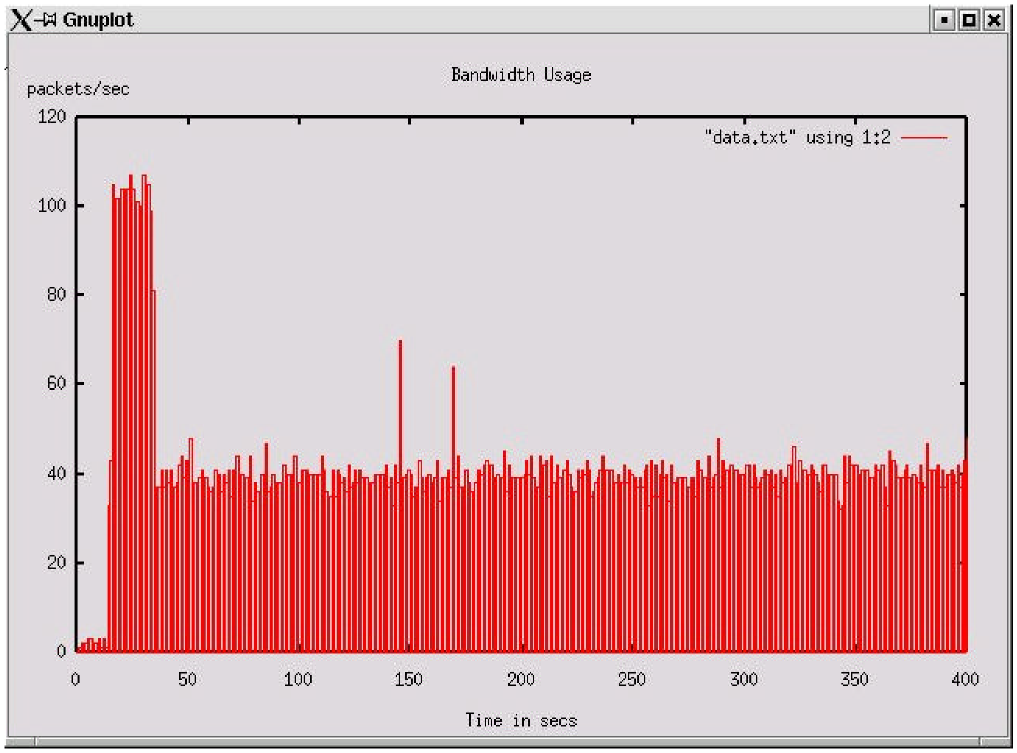


Figure 5.6 – QoS Experienced by A2D2 Client During Non-Stop ICMP DDoS Attack with A2D2 Firewall Policy & CBQ Enabled

To ensure that A2D2 can mitigate a large variety of DDoS attacks known to date, a TCP-SYN DDoS attack was launched with Stacheldraht against A2D2 with Firewall Policy and CBQ enabled. The A2D2 Client QoS was evaluated. Figure 5.7 did not indicate severe QoS degradation but showed slight fluctuations in packet reception. The streamed video was played continuously without freezing or timing-out. However, the screen quality was noticeably inferior by observation. For example, a couple of frames in a screen shot might be frozen while the rest of the screen refreshed. RealPlayer statistics showed that a total of 22,179 packets were received for the 10-minute video clip, slightly smaller than the baseline total of around 23,000 packets. A large number of retransmission requests were sent out (4,090 requests). At the same time, a large number of packets are recovered (2,641 packets) leaving about 1,449 lost packets.

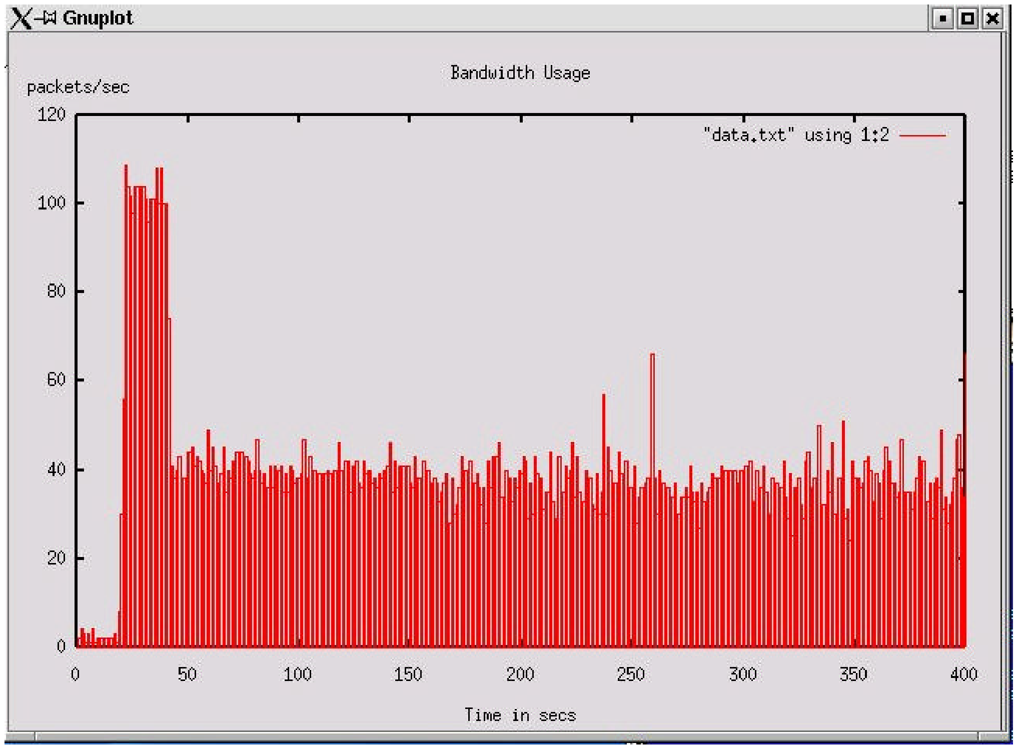


Figure 5.7 – QoS Experienced by A2D2 Client During Non-Stop TCP-SYN DDoS Attack with A2D2 Firewall Policy & CBQ Enabled

It is unclear why the TCP-SYN DDoS attack launched by Stacheldraht would result in slight degrades of QoS for a network with a firewall policy and CBQ enabled. One explanation could be that more TCP ports are opened to provide such services as telnet and ssh. Nine TCP ports were opened in the firewall policy while only four UDP ports were configured to accept packets. Another reason could be that TCP-SYN attacks consume the kernel data structure of the RealServer. Information about the

SYN packets is stored in the data structure until the connection requests are closed by the SYN-ACK packets. Since DDoS SYN attack packets contained spoofed addresses, the connection requests remain open and continue to occupy the kernel data structure. As a result, the RealServer might slow down its response to the RealPlayer requests.

To test the intrusion tolerance of the A2D2 network, the final test scenario enabled all features of the A2D2 network, including firewall policy, CBQ, Snort IDS detection, autonomous alert communication between IDS and firewall and multi-level rate-limiting. Figure 5.8 showed that the fully equipped A2D2 network further improved QoS during TCP-SYN attacks. A total of 23,444 packets were received by the A2D2 RealPlayer Client similar to those received by the baseline test scenario.

Although there was no observable service degrades, RealPlayer indicated that there were 49 retransmission requests of which 40 packets were recovered and nine were lost. The test was repeated five times and each showed similar results where the total packets received are similar, no service degrades were observed and the graphical presentation of the packet reception activity at RealPlayer Client resembled that of the baseline. However, in each run, there are retransmission requests and packet recovery. The number of retransmission requests varied from 49 to 1376 and packet recovery rate ranged from 40 to 776 while around 9 to 600 were lost. Despite the retransmission requests, the QoS is significantly improved in the fully featured A2D2 network compared to the network where only firewall policy and CBQ were implemented. The number of retransmission requests is considerably fewer in the fully featured A2D2 network than the policy-CBQ only network that consistently sent out more than 4000 retransmission requests. The number of packets lost by the fully featured A2D2 network recorded was more than 1,000 packets fewer than that of the policy-CBQ network.

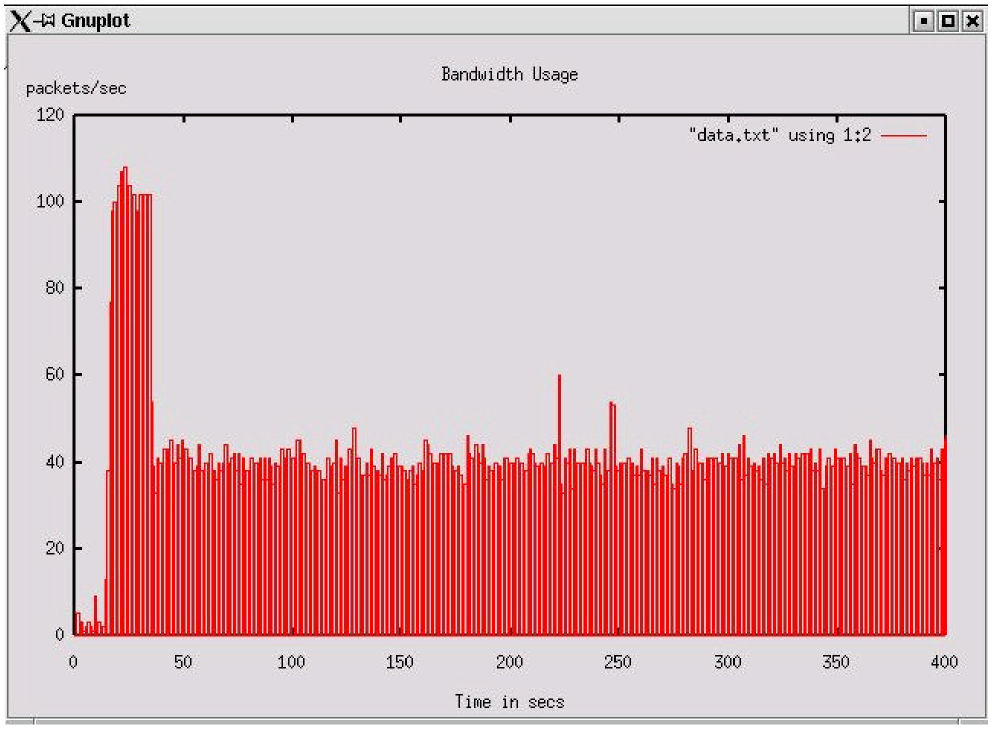


Figure 5.8 – QoS Experienced by A2D2 Client During Non-Stop TCP-SYN DDoS Attack with A2D2 Firewall Policy, CBQ, Snort IDS, Autonomous Alert Communication and Multi-level Rate-Limiting Enabled

One observation is that the number of retransmission requests and packets lost experienced by clients of the fully featured A2D2 network depend on the delay time between the start of the attack and when rate-limiting rules were applied. At some test runs, rate-limiting rules were applied immediately after Snort IDS sent out the alerts.

In other cases, the IDS alerts seem to have problems connecting to the firewall gateway computer (Titan). A number of "Failed to Connect to Rate-Limiter" messages were observed. As mentioned in Section 5.3, the firewall gateway computer may "hang" during an TCP-SYN attack, most probably due to the lack of processing power in handling the various resource-intensive processes. While the firewall gateway may not respond to keyboard or mouse inputs, it consistently accepted IDS alerts and applied rate limiting rules accordingly even though rule application may be delayed. In every test run, after the attack was stopped, the iptables status was checked using the command "iptables –v –L". A representative output of the iptables status is presented in Table 5.2.

As shown by Table 5.2, a large number of packets were rate-limited or dropped in each of the rate-limiting levels L3, L2, and L1. The IDS identified the subnet flood 128.198.61.0/24. The rate-limiter identified packets coming from this subnet and applied rate-limiting rules accordingly. At the completion of the test run, approximately 6,523,000 packets with 261,000,000 bytes coming from 128.198.61.0/24 were passed to rate-limiting level 1 from the INPUT, FORWARD, and OUTPUT chains and were dropped. Before level 1 rate-limiting was applied, about 33,505 packets were passed to rate-limiting level 3 of which 33,234 were dropped and 271 were accepted. Approximately 1,159 packets were marked level 2 of which 1,132 were dropped and 27 were accepted.

Additional evidence to prove the effectiveness of the multi-level rate-limiting technique is shown by the A2D2 RealServer (Pluto) statistics presented in Table 5.1.

During normal network activities, the average incoming rate of packets is 3.13 packets per second. During DDoS attacks, the average incoming rate of packets ranges from 31.19 packets per second to 505.25 packets per second depending on the duration of the attack, the type of attack and mitigation techniques employed. The fully featured A2D2 recorded an average incoming rate of 10.87 packets per second, the closest to the baseline value. The 10.87 packets-per-second-rate is especially noteworthy when compared to the 299.34 packets per second rate experienced by the RealServer during an identical TCP-SYN attack with just CBQ and policy enabled.

```
Chain INPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
6336K  253M level1     all  --  any    any     128.198.61.0/24      anywhere
    0     0 ACCEPT     udp  --  any    any     anywhere             anywhere            udp dpt:domain
2760K  111M ACCEPT     all  --  any    any     anywhere             anywhere

Chain FORWARD (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
 186K 7454K level1     all  --  any    any     128.198.61.0/24      anywhere
 139K   60M ACCEPT     all  --  eth1   any     anywhere             anywhere
88701 3630K ACCEPT     all  --  any    any     anywhere             anywhere

Chain OUTPUT (policy DROP 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    2   142 ACCEPT     udp  --  any    any     anywhere             anywhere            udp dpt:domain
2742K  110M ACCEPT     all  --  any    any     anywhere             anywhere

Chain level0 (0 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DROP       all  --  any    any     anywhere             anywhere

Chain level1 (2 references)
 pkts bytes target     prot opt in     out     source               destination
6523K  261M DROP       all  --  any    any     anywhere             anywhere

Chain level2 (0 references)
 pkts bytes target     prot opt in     out     source               destination
   27  1080 ACCEPT     all  --  any    any     anywhere             anywhere          limit: avg 50/sec burst 5
 1132 45280 DROP       all  --  any    any     anywhere             anywhere

Chain level3 (0 references)
 pkts bytes target     prot opt in     out     source               destination
  271 10840 ACCEPT     all  --  any    any     anywhere             anywhere          limit: avg 151/sec burst 5
33234 1330K DROP       all  --  any    any     anywhere             anywhere
```

Table 5.2 – iptables Status After A2D2 Implementation (iptables –v –L)

Chapter 6

A2D2 Test-bed Limitations and Future Work

6.1 Firewall Processing Speed

One of the main limitations of the A2D2 test-bed is the processing speed of the firewall computer (Titan). This computing power limitation poses two important issues as explained in Sections 5.3 and 5.4:

- The accuracy of the IPTraf log data for the two network interfaces of the firewall computer (Titan).
- The delay between the time when a flood alert is raised and the time when the rate-limiting rule is applied.

Although the effectiveness of the A2D2 network can be inferred from information collected from the A2D2 RealPlayer Clients and the A2D2 RealServer

(Pluto), the IPTraf information from the firewall (Titan) can provide a complete picture of all activities related to the operations of the A2D2 network.

Perhaps a more important reason to upgrade the processing power of the firewall computer is the potential QoS enhancements the A2D2 can provide to its clients. The sooner the rate-limiting rules are applied, the faster attack packets are blocked at the firewall gateway and the better the A2D2 can serve its clients within the scope of available bandwidth. As mentioned in Section 5.4, the number of retransmission requests can range from 49 to over 1000 depending on when the rate-limiting rules are applied. Therefore, it will be beneficial to re-run the tests using a faster computer as the firewall gateway in the A2D2 network.

In addition, the current test-bed did not measure the actual delay time between when an alert was raised and when the rate-limiting rules were applied. The delays are observed on screens. When the IDS sent an alert to the firewall, the alert was printed on the screen. Similarly, screen output was provided when the rate-limiting rules were applied. To facilitate better empirical analysis, future researchers should resort to mathematical means to measure and calculate the actual rate-limiting response time.

6.2 Alternate Route Technique

Since the speed at which rate-limiting rules is applied directly impacts the QoS experienced by the A2D2 clients, it is worthwhile to explore various techniques that may facilitate communications between the IDS and firewall. Currently, the IDS often reported a "Failed to Connect to Rate-Limiter" message during DDoS attacks. Based on such messages and simple observation, in addition to the processing power of the firewall computer, another possible reason is that the network link between IDS and firewall may be congested by attack traffic. Therefore, it is important to explore techniques and designs where a separate "reserved" link is used for communications among A2D2 hosts during an attack.

The "reserved" link idea can be expanded to include a private network link that is used for communications between the A2D2 DMZ and the A2D2 clients during an attack. Such a technique requires multi-paths capability where packets arrive at the A2D2 from the A2D2 clients through the gateway computer but response packets are routed through the backend private links following a different path.

6.3 TCP-SYN Attack

While ICMP and UDP bandwidth consumption attacks can be completely countered by the A2D2 firewall policy and CBQ, TCP-SYN attack requires autonomous multi-level rate limiting to achieve a similar level of QoS. Section 5.4 hypothesizes that the mitigation requirements and results difference is due to the nature of the TCP-SYN attack. While ICMP and UDP are attacks that strictly congest a network link by consuming all available bandwidth, a TCP-SYN attack includes an extra element of overloading the server's kernel connection session data structure. The intrusion tolerance capability of A2D2 can be further verified by studying additional DDoS attacks of various types and natures using the A2D2 test-bed.

6.4 Scalability

The current study of the A2D2 test-bed focuses on the QoS provided by the streaming video RealServer. However, the A2D2 CBQ configuration has taken into account of the common services available in a generic DMZ set-up, such as telnet, ssh, www, and smtp services. Additional tests can be performed to fine-tune the intrusion tolerance of the A2D2 network with respects to other services. Three A2D2 clients were used in the existing A2D2 implementation. The number of A2D2 clients can also be increased for future studies.

6.5 Anomaly Detection

The study of anomaly detection is beyond the scope of this thesis. Nonetheless, flood detection with the Snort IDS is a critical component of the A2D2 design. The current A2D2 design provides administrators with a flexible way to define the flood thresholds for their specific network. Adding an anomaly detection feature to the flood detection will be one valuable enhancement to the A2D2 and can assist administrators in determining a more accurate flood threshold.

6.6 Fault Tolerant

The design of the A2D2 network targets the QoS aspect of intrusion tolerance. Section 2.4.1 also introduced another key area of intrusion tolerance: fault tolerant. The current A2D2 design does not incorporate fault tolerant principles of replication and duplication. Intruders can potentially aim at shutting down the A2D2 firewall gateway or the IDS host, thereby rendering the autonomous nature of the A2D2 network futile. The A2D2 design can be significantly refined by encompassing fault tolerant features.

Chapter 7

Software Engineering Model and Lessons Learned

7.1 The Evolutionary Software Life Cycle Model

The development of the A2D2 network follows the Evolutionary model of the Software Life Cycle Processes as defined by the “Information Technology – Guide for ISO/IEC 12207 [ISO95]. The Evolutionary model develops a system in builds. Unlike the Incremental model that develops systems in a series of builds with defined requirements, the Evolutionary model acknowledges that requirements are not fully understood and cannot be defined initially. Development starts immediately even if the initial requirements are vague and partially defined. Requirements are refined in successive builds as the system evolves [Pig96].

The Evolutionary model, as illustrated in Appendix B [ISO95], is particularly suitable for a research project such as this thesis. At the start of a research project, researchers often explore many knowledge areas of a research domain before deciding on the specific topic. During the course of a research, the research will often change directions due to new insights of the subject areas or unforeseen obstacles.

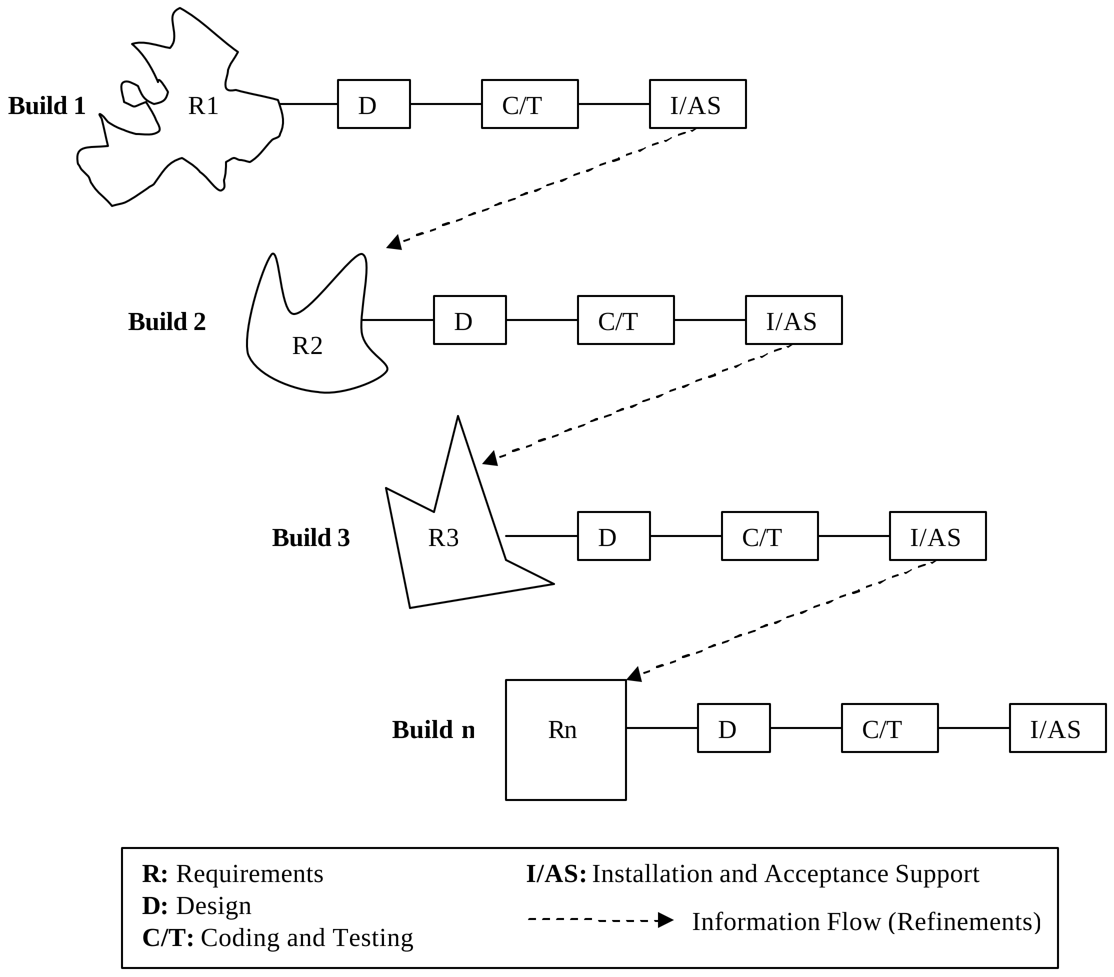


Figure 7.1 – ISO/IEC 12207 (Software Life Cycle Processes) Evolutionary Model

The design and development of the A2D2 network has gone through roughly five builds. Concrete and definite thesis requirements were elucidated only at the end of Build 3, at which point the thesis proposal was presented to the committee members for approval. The approved requirements then served as the base requirements for Build 4. The five builds are:

- Build 1 – Initial Linux Network Test-bed Set-up
- Build 2 – Initial Security Test-bed Design
- Build 3 – Intermediate Security Test-bed Design
- Build 4 – A2D2 Initial Design
- Build 5 – A2D2 Refined Design

7.1.1 Build 1 – Initial Linux Network Test-bed Setup

The process of requirements gathering was started at the end of January 2002, approximately four months before Build 1 was initiated on May 27, 2002. Initial requirements gathering provided a thorough overview of the problem domain and identified that a more in-depth study of the problem domain was required before further analysis. Therefore, the goal of Build 1 was to set up a basic Linux network so that DDoS scenarios can be studied. Instead of an anti-DDoS network, Build 1 focused on the set-up of a Linux network that could route traffic from a private subnet to the public domain. The initial Linux Network Test-bed set-up is presented in Appendix B. The main lesson learned during this build was Linux router configuration. IP Forwarding and Masquerading are enabled through the Network Address Translation table (nat) and the following steps are taken:

- iptables --table nat --append POSTROUTING --out-interface eth0 -j MASQUERADE
- iptables --append FORWARD --in-interface eth1 –j ACCEPT
- echo 1 > /proc/sys/net/ipv4/ip_forward

7.1.2 Build 2 – Initial Security Test-bed Design

Build 2 investigated a test-bed design that facilitates the study of a simple DoS attack from one host. The design is provided in Appendix B. In addition to the behavior of DoS, the network also enables the study of some DoS mitigation techniques such as rate limiting. One main obstacle encountered during this build is the testing of the firewall policies and rate-limiting rules. Considerable time was spent to find out why the rate-limiting rules for SYN flood were ineffective while the rules for ICMP flood produced accurate results. A simple SYN flood was started by "namp –sS –0 –P0 hostname" while an ICMP flood was executed by "ping –f". Rate limiting rules for TCP are applied as follows:

- iptables - A INPUT -p tcp --syn -m -limit --limit 1/s --limit_burst 5 -j ACCEPT
- iptables - A INPUT -p tcp DROP

Rate limiting rules for ICMP are applied as follows:

- iptables - A INPUT -p icmp --syn -m -limit --limit 1/s --limit_burst 5 -j ACCEPT
- iptables - A INPUT -p icmp DROP

When the “ping” attack was stopped, the command line displayed the statistics related to the attack including the percentage of packets lost and received. Ethereal was used to capture packet information to verify whether the SYN rate-limiting rules were effective. However, analysis on ethereal data showed that all packets targeted at the host were received.

It was later determined that the testing methodology was flawed. Ethereal operated on the physical level and displayed statistics of all incoming packets that reached the physical layer. The firewall policies and rate limiting rules were executed by the “iptables” command that operated on the IP layer. Therefore, information for all packets was captured by ethereal before the rate limiting and firewall rules were applied. To accurately evaluate firewall and rate limiting rules, the network-monitoring tool IPTraf was used to monitor traffic from both the external and internal network interfaces, eth0 and eth1. IPTraf was used continuously for future builds.

7.1.3 Build 3 – Intermediate Security Testbed Design

The requirements of Build 3 were to extend Build 2 capabilities so that the behavior of DDoS could be evaluated and additional mitigation techniques could be examined. The design of the Intermediate Security Test-bed is presented in Appendix B. Stacheldraht was chosen to be the DDoS attack tool used in the test-bed and other

DoS tools such as ipsyn, udpf were also evaluated. In addition, various IDS were examined and Snort was chosen to be the detection component of the test-bed. Additional DDoS mitigation techniques such as CBQ were also explored.

Due to the experience gained from the flawed testing methodology used during Build 2, special attention was paid to the networking layer at which each test-bed components operated. At Build 3, the IDS Snort was placed at the firewall gateway computer. It was then determined that the Snort IDS analyzes packets at the physical layer while the firewall rules were applied at the IP layer. As a result, the IDS would continue to raise intrusion alerts even when appropriate rate limiting or firewall rules were applied to block the particular intrusion. Therefore, a more efficient design was needed to incorporate the various components of the security test-bed.

7.1.4 Build 4 – A2D2 Initial Design

Starting at Build 4, the requirements analysis was shifted from the problem domain to the solution domain. Based on the lessons learned from the previous builds, the appropriate mitigation channels were identified, the required software development and improvements were defined and the A2D2 design was introduced. The initial A2D2 design is illustrated in Appendix B. The dominant processes in Build 4 were the design, coding and testing phases as opposed to the previous builds where requirements analysis, installation and testing were the key processes. Some major accomplishments at this build include the design of the A2D2 network, the set-up of the A2D2 DMZ, and the design and implementation of the Snort flood detection preprocessor.

7.1.5 Build 5 – A2D2 Refined Design

One promising feature of the A2D2 network identified at Build 4 was the provision of a private alternate route that could substitute the front-end gateway during a DDoS attack. However, the amount of work involved in the design and implementation of the Snort flood detection preprocessor required the thesis scope to be scaled down. Build 5 also focused on the design and implementation of multi-level rate limiting as well as the autonomous cooperation among all A2D2 components. The refined A2D2 design, shown in Appendix B, has demonstrated a high level of intrusion tolerance during various types of DDoS attacks.

Chapter 8

Conclusion

Many advances have been made by security professionals in the past few years to counter the mass disruptions created by powerful DDoS tools. Nonetheless, the number of DDoS victims continues to increase every year as new DDoS tools are easily shared among curious intruders. Accepting the fact that it is still impossible to completely stop or prevent DDoS attacks, many researchers have focused on intrusion tolerance techniques that maximize quality of service during DDoS attacks. Most promising intrusion tolerance research has taken a macro approach that recruits such major players as Internet Service Providers (ISP) and protocol designers to adopt a collaborative intrusion trace back, detection and push back mechanism. While researchers are searching for an ultimate global solution, a vast number of individual Internet users become DDoS victims. These users of small and medium sized networks often lack the influence and knowledge to set up an elaborate collaborative infrastructure. Nor do these SOHO users have the incentive to spend thousands of dollars on commercial products that are designed to alleviate this one type of intrusion.

The design of A2D2 provides a micro approach where individual Internet users can address the DDoS problems in relatively manageable and affordable ways. The A2D2 network allows SOHO users to utilize existing and free technologies and to take control of their own defense within their own network boundary. A2D2 effectively combines firewall policy, CBQ, multi-level rate-limiting and DDoS flood detection in an autonomous architecture. Test-bed results clearly show that the A2D2 design demonstrates tolerance against bandwidth consumption attacks of various types, including UDP, ICMP and TCP-based DDoS attacks. When one type of mitigation technique, such as a firewall policy, fails to completely handle an attack, another technique such as CBQ or rate-limiting will automatically provide added protection. Regardless of the types of DDoS attacks, A2D2 clients enjoy QoS similar to the level of service they experience during normal network activities.

To further improve the intrusion tolerance of the A2D2 network, fault tolerant principles of duplication and diversification of services should be applied. Additional features can be added to enable alternative routes should the gateway link become congested. Anomaly flood detection can further assist administrators in configuring the A2D2 network. It is hopeful that with continual enhancements, the proposed A2D2 network design can be deployed in mass numbers among small and medium-sized networks, thereby improving the overall Internet security against DDoS bandwidth consumption attacks.

Bibliography

[ASTA00] Astalavista Network Library Archive. http://www.astalavista.com/archive/index.asp?dir=ddos

[AV02] Arul Anand & Vanitha. QoS Implementation in Linux - A White paper. ViSolve.com. Feb 04,2002. http://squid.visolve.com/white_papers/qos.htm

[BLG$^+$99] Andrew Barkley, Steve Liu, Quoc Thong Le Gia, Matt Dingfield and Yashodhan Gokhale. A Testbed for Study of Distributed Denial of Service Attacks (WA 2.4). Proceedings of the 2000 IEEE Workshop on Information Assurance and Security. United States Military Academy, West Point, NY, 6-7 June, 2000.

[BLT02] Steve Bellovin, Marcus Leech, and Tom Taylor. ICMP Traceback Messages. Internet Draft: draft-ieft-itrace-01.txt. Expires April 2002

[CERT98] CERT Coordination Center. “Determine contractor ability to comply with your organization's security policy”. http://www.cert.org/security-improvement/practices/p019.html

[CERT01] CERT Coordination Center. Denial of Service Attacks. http://www.cert.org/tech_tips/denial_of_service.html

[Cis99] Cisco Systems. Unicast Reverse Path Forwarding. Cisco IOS Documentation, May 1999.

[Cis02] Cisco Systems. Policing and Shaping Overview. Cisco IOS Release 12.0 Quality of Service Solutions Configuration Guide. http://www.cisco.com/univercd/cc/td/doc/product/software/ios120/12cgcr/qos_c/

[Cis02-2] Cisco Systems. White Paper. Building a Perimeter Security Solution with the Cisco Secure Integrated Software. September 6, 2002. http://www.cisco.com/warp/public/cc/pd/iosw/ioft/iofwft/tech/firew_wp.htm

[CNW$^+$99] H.Y. Chang, R. Narayan, S.F. Wu, B.M. Vetter, X. Wang, M. Brown, J.J. Yuill, C. Sargor, F. Jou, and F. Gong. Deciduous: Decentralized Source Identification for Network-Based Intrusion. 6th IFIP/IEEE International Symposium on Integrated Network Management, IEEE Communications Society Press, May 1999. http://www.silicondefense.com/research/itrex/archive/tracing-papers/chang99deciduos.pdf

[Comer00] Douglas Comer. Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture (4th Edition). Prentice Hall. January 18, 2000

[Comp] Computer Networking Glossary. http://compnetworking.about.com/library/glossary/bldef-qos.htm?terms=QoS

[CR00] Matt Curtin and Marcus J. Ranum. Internet Firewalls: Frequently Asked Questions. December 1, 2000

[CSI02] Computer Security Institute. “Cyber Crime Bleeds U.S. Corporations, Survey Shows; Financial Losses from Attacks Climb for Third Year in a Row”. http://www.gocsi.com/press/20020407.html

[Des02] Paul Desmond. Cisco, Enterasys Deliver New IDS Products. boston.internet.com. May 16. 2002. http://boston.internet.com/news/article.php/2371_1135921

[Dit99] David Dittrich. The "stacheldraht" distributed denial of service attack tool. The DoS Project's "trinoo" distributed denial of service attack tool. The "Tribe Flood Network" distributed denial of service attack tool The "mstream" distributed denial of service attack tool http://www.washington.edu/People/dad/

[Dit00] David Dittrich. “Usenix Security Symposium 2000 DDoS – Is there Really a Threat”. http://staff.washington.edu/dditrich/talks/sec2000/

[Dit02] David Dittrich. Distrib uted Denial of Service Attacks/Tools. http://staff.washington.edu/dittrich/misc/ddos/

[For01] Jeff Forristal. Fireproofing Against DoS Attacks. Network Computing. December 10, 2001. http://www.networkcomputing.com/1225/1225f3.html

[Fred02] Karen Kent Frederick. Evaluating Network Intrusion Detection Signatures. Part I. SecurityFocus Online. September 10, 2002.

[FS00] P. Fergusion and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks Which Employ IP Source Address Spoofing. RFC 2827. May 2000.

[Gnuplot] Gnuplot Central. http://www.gnuplot.info/

[GWW+00] Katerina Goseva-Popstojanova, Feiyi Wang, Rong Wang, Fengmin Gong, Kalyanaraman Vaidyanathan, Kishor Trivedi, Balamurugan Muthusamy. Charaterizing Intrusion Tolerant Systems Using a State Transition Model. Duke University, MCNC Research Triangle Park, Vitesse Corp. 2000.

[HMM+02] Bert Hubert, Gregory Maxwell, Remco van Mook, Martijn van Oosterhout, Paul B Schroeder, and Jasper Spaans. Linux Advanced Routing & Traffic Control HOWTO. DocBook Edition. http://www.kiarchive.ru/pub/linux/docs/HOWTO/Adv-Routing-HOWTO

[HMP+01] Allen Householder, Art Manion, Linda Pesante, George M. Weaver, Rob Thomas. Managing the Threat of Denial-of-Service Attacks. CERT Coordination Center. October 2001. http://www.cert.org/archive/pdf/Managing_DoS.pdf

[IANA02] Internet Assigned Numbers Authority. http://www.iana.org/assignments/port-numbers

[IB02] John Ioannidis and Steve M. Bellovin. Implementing Pushback: Router-Based Defense Against DDoS Attacks. AT & T Research Lab. 2002. http://citeseer.nj.nec.com/ioannidis02implementing.html

[IET00] IETF – Internet Engineering Task Force. Request for Comments: 2827. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. May 2000. http://www.ietf.org/rfc/rfc2827.txt.

[IET02] IETF – Internet Engineering Task Force. ICMP Traceback (itrace) Charter. Last modified: March 28, 2002. http://www.ietf.org/html.charters/itrace-charter.html

[ISO95] ISO/IEC. 1995. Information Technology – Guide for ISO/IEC 12207 (Software Life Cycle Processes). Draft Technical Report.

[ITW01] ITWorld.com. CERT hit by DDoS attack for a third day. May 24, 2001. http://www.itworld.com/Sec/3834/IDG010524CERT2/

[KMW01] Frank Kargl, Joern Maier, and Michael Weber. "Protecting Web Servers from Distributed Denial of Service Attacks". University of Ulm Germany, May 2001. http://citeseer.nj.nec.com/444367.html

[Kum95] Sandeep Kumar. Classification and Detection of Computer Intrusions. Ph.D. Dissertation. Purdue University. August 1995.

[Lai00] Brian Laing. How To Guide: Intrusion Detection Systems. Internet Security Systems. http://gd.tuwien.ac.at/infosys/security/oldsnort/docs/iss-placement.pdf

[Lin02] "Network Intrusion Detection Using Snort" http://www.linuxsecurity.com/feature_storeis/feature_story-49.html

[MMW+02] Dan Massey, Allison Mankin, C. L. Wu, X. L. Zhao, Felix Wu, W. Huang, and Lixia Zhang. Intention-Driven ICMP Trace-Back. Internet Draft: draft-ietf-itrace-intention-00.txt. Expires May 2002.

[MVS01] David Moore, Geoffrey M. Voelker and Stefan Savage. Inferring Internet Denial-of-Service Activity. 2001 http://www.cs.ucsd.edu/~savage/papers/UsenixSec01.pdf.

[NCFF01] Stephen Northcutt, Mark Cooper, Matt Fearnow, Karen Frederick. Intrusion Signatures and Analysis. Indiana: New Riders. January 2001.

[NIST95] National Institute of Standards and Technology. A Conceptual Framework for System Fault Tolerance. 1995. http://hissa.nist.gov/chissa/SEI_Framework/framework_1.html

[Nmap02] Nmap: Network Mapper. http://www.insecure.org/nmap/index.html.

[OBSD02] OpenBSD. Using IPsec (IP Security Protocol). http://www.openbsd.org/faq/faq13.html#What.

[Pig96] Thomas M. Pigoski. Practical Software Maintenance: Best Practices for Managing Your Software Investment. NY: Wiley Computer Publishing. 1996.

[RL93] Y. Rekhter and T. Li. RFC 1518: An Architecture for IP Address Allocation with CIDR. September 1993.

[RS91] Deborah Russell and G. T. Gangemi Sr. Computer Security Basics. O'Reilly & Associates, Inc., Sebastopol, California, December 1991.

[San00] Scott C. Sanchez. IDS Zone Theory Diagram. 2000. http://www.snort.org/docs/scott_c_sanchez_cissp-ids-zone-theory-diagram.pdf

[SANS00] SANS Institute – Global Incident Analysis Center. Egress Filtering v 0.2. http://www.sans.org/y2k/egress.htm

[Sao02] Greg Saoutine et al. Barbarians at the Gate. Microsoft Professional Magazine. September 29, 2002.

[SDW+01] Dan Sterne, Kelly Djahandari, Brett Wilson, Bill Babson, Dan Schnackenberg, Harley Holliday, and Travis Reid. "Autonomic Response to Distributed Denial of Service Attacks". Recent Advances in Intrusion Detection. 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, 2001 Proceedings. http://www.cse.ogi.edu/~wuchang/cse581_winter2002/papers/22120134.pdf

[Snort] Snort. The Open Source Network Intrusion System. http://www.snort.org.

[Stal99] William Stallings. Cryptography and Networks Security: Principles and Practice. Second Edition. New Jersey: Prentice Hall. 1999.

[Whatis] Whatis?com. http://whatis.techtarget.com/definition/

[Whi00] Whisker. http://sourceforge.net/projects/whisker/

[Whit01] Whithats, Inc. arachnids – The Intrusion Event Database. IDS246/DOS-LARGE-ICMP.

http://www.whitehats.com/cgi/arachNIDS/Show?_id=ids246&view=event

[WO01] Jeroen Wortelboer and Jan Van Oorschot. Linux Firewall – the Traffic Shaper. January 15, 2001. http://online.securityfocus.com/infocus/1285

[YEA00] Jianxin Yan, Stephen Early, Ross Anderson. “The XenoService – A Distributed Defeat for Distributed Denial of Service. Proceedings of ISW 2000. http://citeseer.nj.nec.com/376606.html

[ZOS00] Weibin Zhao, David Olshefski and Henning Schulzrinne. Internet Quality of Service: an Overview. Columbia University. 2000. http://citeseer.nj.nec.com/zhao00internet.html

Appendix A Flood Preprocessor Logic Flowchart

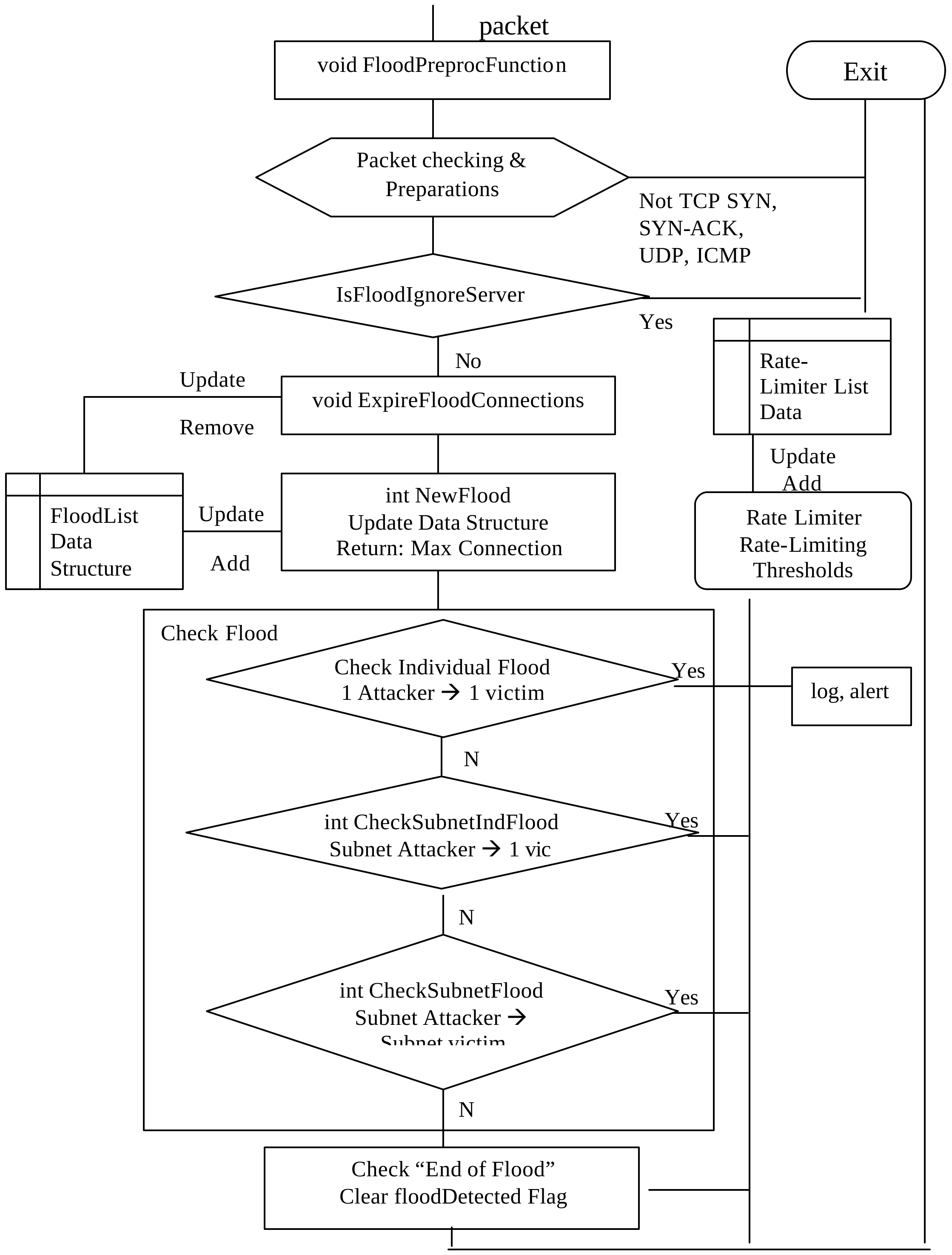


Figure A.1 – Flood Preprocessor Logic Flowchart

Appendix B Design Evolution

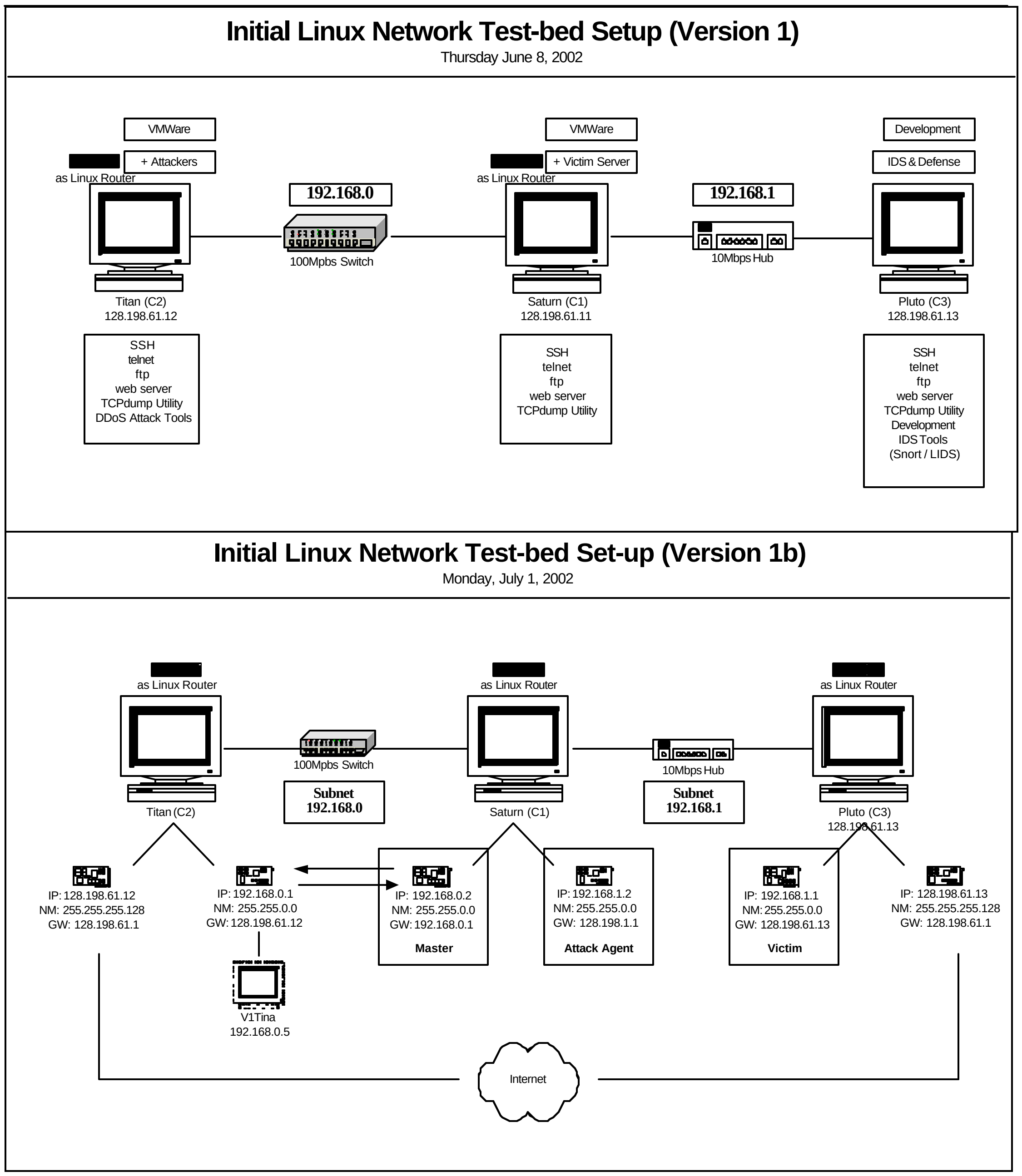


Figure B.1 – Build 1 – Initial Linux Network Test-bed Set-up

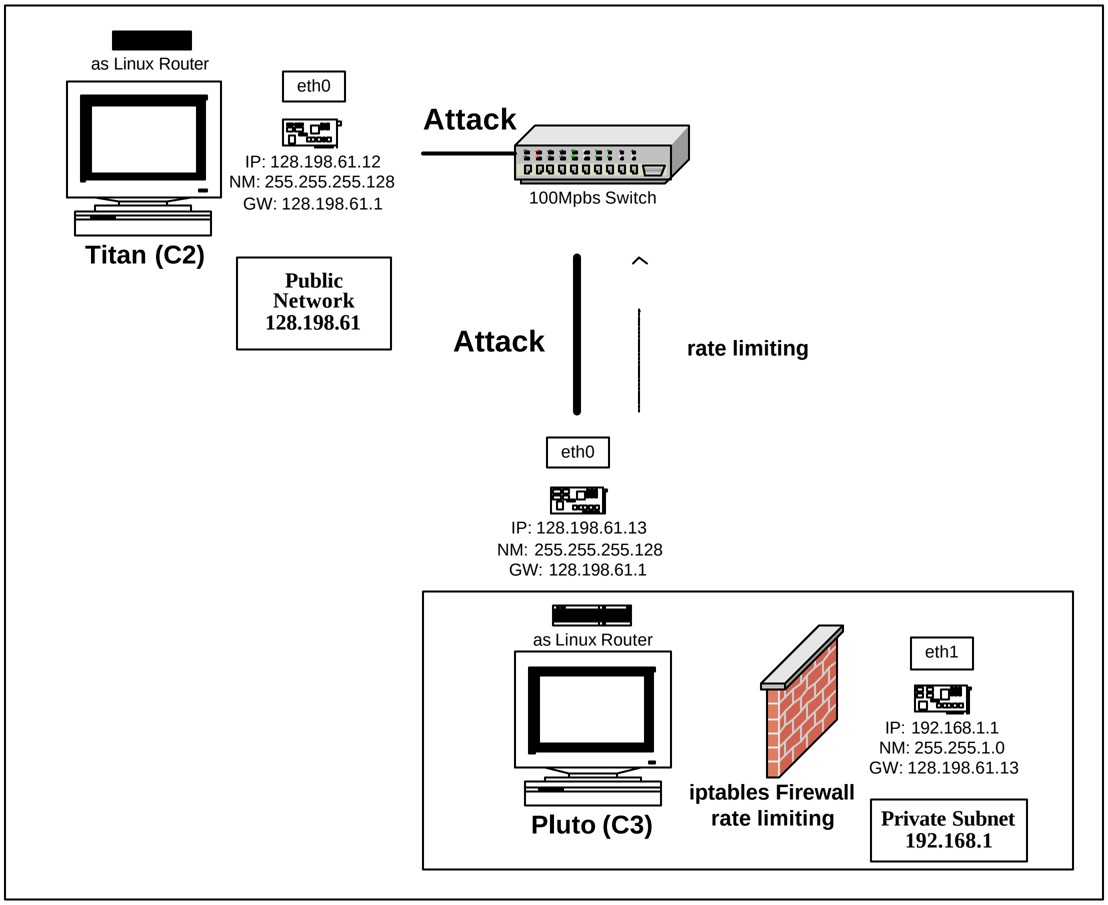


Figure B.2 – Build 2 – Initial Security Test-bed Design

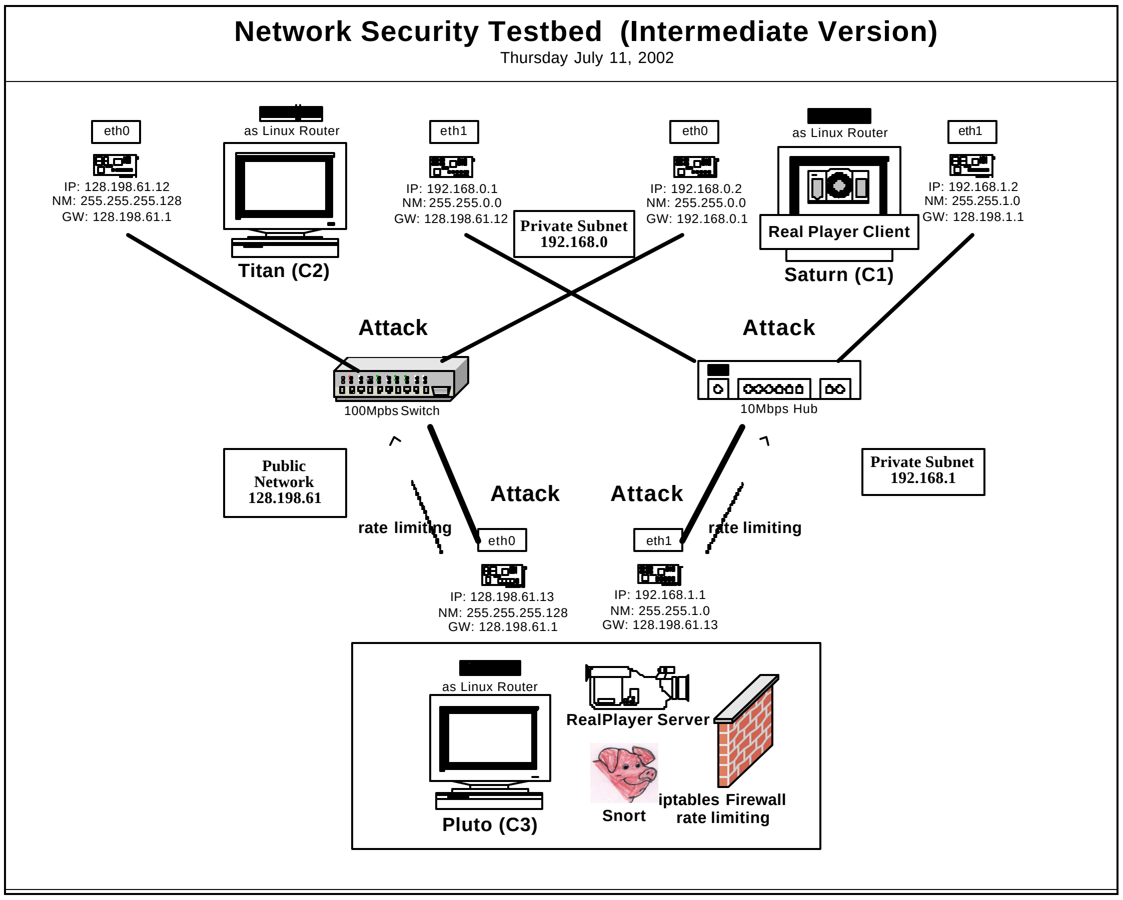


Figure B.3 – Build 3 – Intermediate Security Test-bed Design

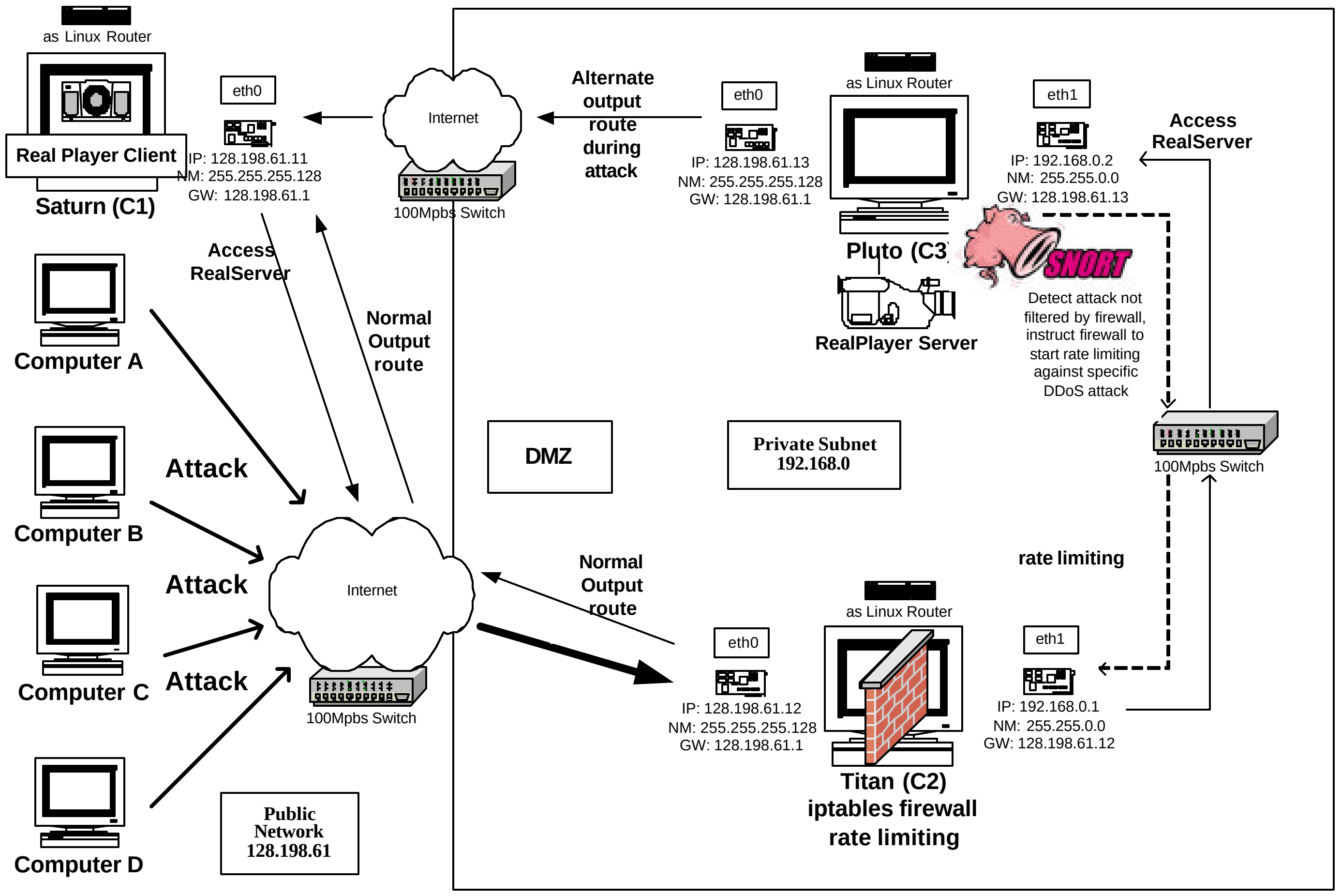


Figure B.4 – Build 4 – A2D2 Initial Design

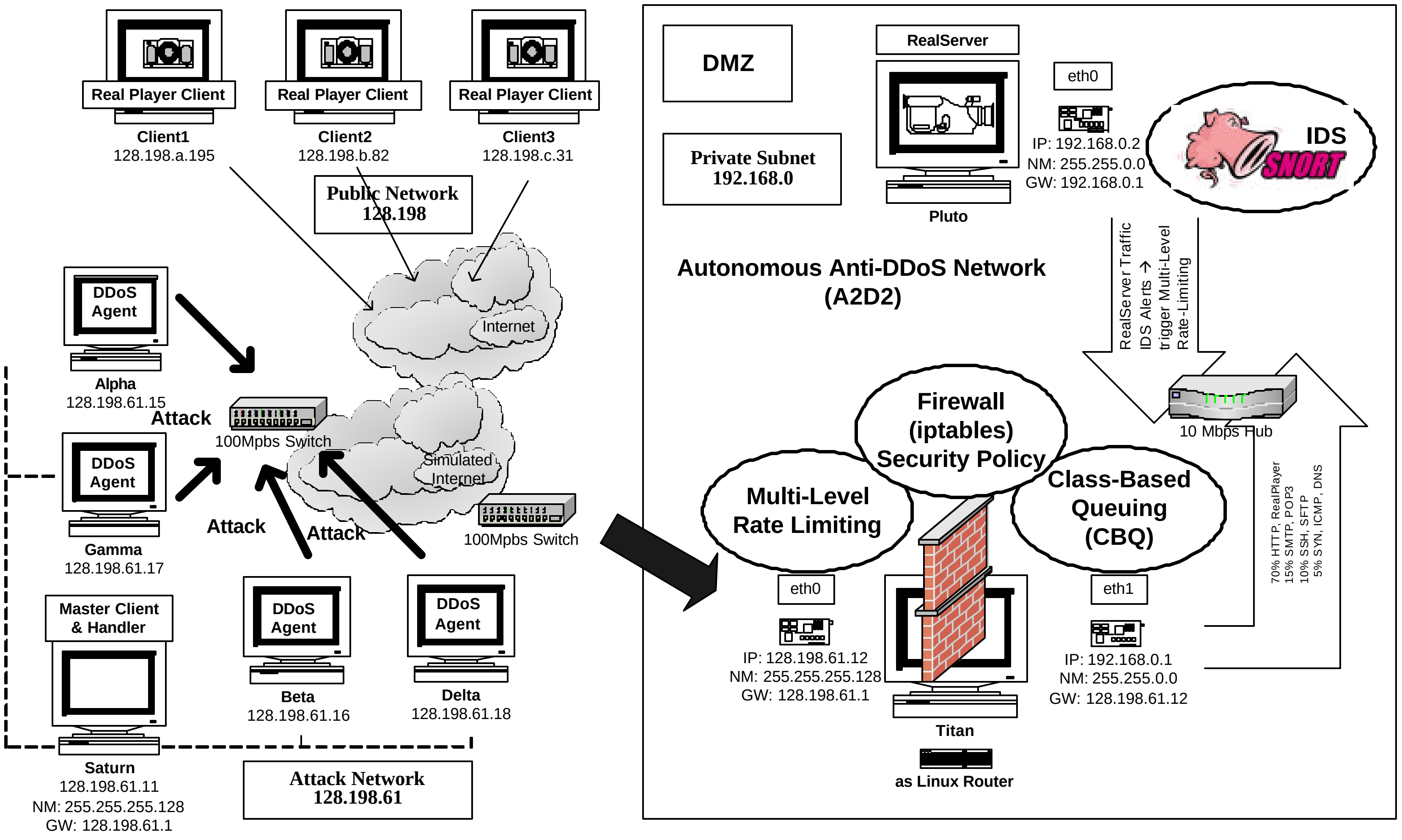


Figure B.5 – A2D2 Final Design

Appendix C A2D2 Users Manual

The A2D2 network is easily configured to guide against various types of DDoS attacks. This users manual explains the steps required to configure and start the A2D2 network. The A2D2 setup is assumed to resemble that illustrated in Appendix Figure B.5. A sample demonstration script is also provided as a step-by-step guide on how a DDoS attack is detected and controlled. The set up of the A2D2 test-bed includes three main components:

- The attack network
- The autonomous defense network (A2D2)
- The client network

C.1 The A2D2 Attack Network

The attack network consists of any number of clients, handlers and agents. Before configurations, users have to first identify the computers involved and their respective roles.

C.1.1 The Essential Software and Files

Users need to identify all essential files related to an attack tool. DDoS tools can be easily located in the Internet [ASTA00]. Each tool contains different files but the file structures are similar among DDoS tools. Users need to identify the files for clients, handlers and agents of their chosen tools. For StacheldrahtV4, the files are:

Client Stacheldrahtv4/telnetc/	**Handler Stacheldrahtv4/**	**Agent Stacheldrahtv4/leaf/**	**Encryption Stacheldrahtv4/**
Bf_tab.h Client.c makefile	Telnetc Bf_tab.h Config.h Makefile Mserv.c Tubby.h	Bf:tab.h Config.h Config.h.in Control.h Icmp.c Makefile Sysn.c Td.c Tubby.h Udp.c	Blowfish.c Blowfish.h
Executable			
Client	***mserv***	***td***	***N/A***

Table C.1 – Stacheldrahtv4 Files

Before compilation, the following steps need to be taken:

1. Ensure the client and handler share the same communication port number
   - in client.c (#define MASTERSERVERPORT 1234)
   - in mserv.c (#define MSERVERPORT 1234)
2. Define the IP address of the handler
   - in mserv.c (#define LOCALIP "5.6.7.8")
3. Define the password for communications with the handler
   - in mserv.c (#define SALT "zAeaLAzwZ7Eng")
   - //encrypted password for "manager"
4. Ensure the agents know who their handler is
   - in td.c (#define MSERVER1 "5.6.7.8")
5. Ensure the handler and agents share the same communication port number

- in mserv.c (#define COMMANPORT 9000)
- in td.c (#define COMMANPORT 9000)

C.1.2 The Execution Steps

1. Place the executable of "Client", "mserv" and "td" on machines that are designated as clients, handlers, and agents respectively.
2. At Client, execute the command "./Client handlerIPaddress".
3. Type in the password "manager" when prompted.
4. Upon successful login, the DDoS manager will indicate the number of "alive" agents and "dead" agents at the command prompt. In an example where there are four "alive" agents and zero "dead" agent, the command prompt <a4! d0!> will be shown.
5. Users can then execute the various command specified in ".help".
6. For example, user can launch a UDP attack against a victim with the following command ".mudp victim.target.edu". Other available attacks are .micmp and .msyn.
7. To end the attack, execute the command ".mstop all".
8. To exit Stacheldraht, type .quit.

C.2 The Autonomous Defense Network (A2D2)

There are three main components in the A2D2 DMZ: the firewall gateway, the IDS and the server that provides video streaming service. In the A2D2 test-bed setup, the IDS and the RealServer reside in the same machine, Pluto. In a more elaborate setup, the IDS and the RealServer should be separated into different hosts. The following sections describe the procedures to configure and execute each of the three components.

C.2.1 The Firewall Gateway (Titan)

C.2.1.1 The Essential Software and Files

The firewall polices and CBQ implementations are contained in three shell scripts:

- noudp-policy.sh
- nocbq.sh
- cbq.sh

The noudp-policy.sh contains simple TCP policies and network address translation (NAT) rules so that packets can be routed from the internal private network to the public domain. The nocbq.sh file implements TCP and UDP policies while the cbq.sh script adds CBQ implementation to the basic firewall policies. In addition to the policy script and the CBQ script, rate-limiting files are contained in the “rateif” directory. Four files are contained within the “rateif” directory that enable autonomous mutli-level rate-limiting.

- rateif.conf
- rateif.pl
- logfile

- rulefile

The “rateif.conf” file is the rate-limiting interface configuration file. Users can define six sets of parameters:

- The number of rate-limiting levels and the rate and expiration times associated with each level. Details of how to configure the various rate parameters in the rateif.conf file are described in Section 4.2.3.
- The server name on which the rate-limiting rules are applied
- The port number at which the rate limiter will listen for the IDS alert
- The filename that log all the rate rules that are being applied
- The location of where the iptables program is at the firewall server
- The alarm time interval at which the rate-limiting rules will be checked for their expiration.

The “ratief.pl” file is the main program that accepts the IDS alert and limits packet rate accordingly. The “logfile” is the file that is being used by the “rateif.pl” program internally to calculate expiration time. The rulefile is the log file that contains all rate limiting rules that are being applied.

C.2.1.2 The Execution Steps

1. Make sure the deprecated ipchains redhat packet manager (rpm) does not exist in the systems. The ipchains rpm conflicts with the iptables rpm if both are installed in the same system.
    - rpm –qa | grep ipchains

- rpm –e ipchains*.rpm

2. Flush all firewall rules to ensure no hidden policies are applied without users' knowledge. This can be achieved by:
   - rebooting the system
   - sh <anyscriptname.sh> stop (e.g. sh cbq.sh stop)
3. Make sure the rate.conf file is configured with values appropriate to the user's network.
4. Run the rate limiting program by executing the command ./rateif.pl
5. Start the firewall script
   - sh <anyscriptname.sh> start (e.g. sh cbq.sh start)
   - It is important not to run the "restart" command (sh <anyscriptname.sh> restart) since the "restart" command will first stop the firewall and flush all chains (rules) including those created by the rateif.pl file.
6. If users have root access and would like to monitor network traffic activities at any interface, the IPTraf program can be run.
   - iptraf

C.2.2 The IDS (Pluto)

C.2.2.1 The essential software and files

The essential files related to the IDS system are the Snort IDS files which can be downloaded from the Snort website [Snort]. The version of Snort deployed in the

A2D2 test-bed illustrated in Appendix Figure B.5 is Snort 1.8.6 for Linux. One of the essential files within the Snort 1.8.6 files provided by Snort is the snort.conf configuration file. Two additional files: spp_flood.c and spp_flood.h, are created for the flood preprocessor.

- snort-1.8.6.tar.gz (include snort.conf)
- spp_flood.c
- spp_flood.h

In addition to the Snort files, a group of files are created to send the Snort alerts to the rate-limiter firewall gateway. These files reside in the “alert” directory and include the following:

- Makefile
- alert
- misc.h
- msock.h
- report.c
- report.o

The main program file is “report.c” where users define the following parameters:

- #define HOSTNAME “the hostname of the firewall gateway computer on which the rate-limiting will applied” (e.g. “titan.uccs.edu”)
- #define PORTNO 6779

- This port number corresponds with the port number defined in the rateif.conf file. The rate limiter listens for the IDS alert at this port number

- #define LOGFILE "/var/log/snort/alert.log"

C.2.2.2 The Execution Steps

1. Extract the Snort files downloaded from the snort website
   - tar –xzvf snort-1.8.6.tar.gz
2. Configure the snort.conf files according to the instructions provided in the snort web site.
   - Specifically, define the network that the IDS will monitor: var HOME_NET <IP address and subnet mask>
3. Place the spp_flood.c and spp_flood.h files in the snort-1.8.6 main directory
4. Integrate the flood preprocessor with the snort base programs as described in detail in Sections 3.1.4.2, 3.1.5 and 3.3.2.
5. Run snort: ./snort –A UNSOCK
6. Change directory to the alert directory and verify that the report.c file is configured with correct parameters for the rate-limiter
7. Compile the report.c program using "Makefile" that creates the output executable: alert
8. Start the alert system with ./alert –h <rate-limiter-firewall-hostname>

C.2.3 The RealServer (Pluto)

C.2.3.1 The essential software and files

The main files for the RealServer host are the Realsystems files downloaded from the RealNetworks website http://www.realnetworks.com/products/. The products provided by RealNetworks are upgraded constantly. Users should refer to the RealNetworks website for information on upgraded products. In the A2D2 test-bed implementation described in Appendix Figure B.5, RealServer 8 is used and its latest user documentation is available at:

- http://service.real.com/help/library/guides/server8/realsrvr.htm

After extraction of the RealServer files, users should verify that such configuration information as port number and password in the "rmserver.cfg" file is correct. Video files being served by the RealServer should be saved in the "Content" directory under the main "Realsystems" directory.

C.2.3.2 The Execution Steps

1. Place video files in the "Content" directory of the "Realsystems" main folder.
2. Change directory to the "Realsystems" main folder and start RealServer:
   - ./Bin/rmserver rmserver.cfg
3. Change directory to the "snort-1.8.6" folder
4. Start snort by the command: ./snort – A UNSOCK

5. Change directory to the “alert” directory placed under the “snort-1.8.6” folder and start the rate-limiting alert interface.
   - ./alert –h <rate-limiter firewall hostname>

C.3 The Client Network

- Any host running the RealPlayer application can be an A2D2 client. However, only Linux hosts can be used to monitor traffic impact during DDoS attacks.

C.3.1 The essential software and files

The latest version of the RealPlayer software can be downloaded from the RealNetworks website: http://www.realnetworks.com/products/. At the Linux clients, the network-graphing files can be installed to illustrate the number of packets received by the client when accessing the RealServer. The network graphing files are contained in the “plot-final-0928” directory:

- input.sh
- plot.pl
- data.txt
- runplot.sh

The main program that enables the graphing of network activities is “plot.pl”. The program “plot.pl” extracts the “number of packets received” data from the “/proc/net/dev” Linux file. The extracted information is saved in the “data.txt” file. The data in the “data.txt” file is then projected by the Linux graphing utility “gnuplot”.

The “input.sh” script contains graphing parameters that “gnuplot” uses, such as the units for the X-axis and Y-axis and the title of the graph. The “runplot.sh” shell script runs the “plot.pl” program. For the “plot.pl” program to execute correctly, users need to ensure that the Linux computer contains the Perl package “time::hires”.

C.3.2 The Execution Steps

1. Start the RealPlayer and access the following link, assuming the gateway computer is “titan.uccs.edu”.
   - http://titan.uccs.edu:8080/ramgen/videoname.rm?usehostname
2. From a Linux host, change directory to “plot-final-0928”
3. Execute the “plot.pl” program using the following command:
   - sh runplot.sh <time-interval-in-sec> <interface> (e.g. sh runplot.sh 1 eth0)
4. From another Linux terminal, change directory to “plot-final-0928”
5. Execute gnuplot:
   - gnuplot input.sh

C.4 A Sample Demo Script

A sample demo script is provided here to demonstrate how a Stacheldraht TCP-SYN flood can be launched, how the A2D2 network responses to the attack and what the QoS experienced by the A2D2 client is. It is assumed that the setup of the A2D2 network is identical to that described in Appendix Figure B.5. IPTraf can be used on any Linux computer at which users have root access. It is recommended that

IPTraf be used in the RealServer computer (Pluto) and the Linux RealPlayer client computer. The steps to be taken at each host are as follows:

C.4.1 At the firewall gateway computer (Titan)

1. Open 2 terminals on Titan
2. At terminal 1, change directory to "final"
3. Verify no firewall policy is active
   - sh cbq.sh stop
4. At terminal 2, change directory to "rateif"
5. Run rate-limiting program
   - ./rateif.pl
6. At terminal 1 "final" directory, execute the CBQ script
   - sh cbq.sh start
7. To terminate rate-limiting, type "Clt+C"
8. To terminate CBQ, execute "sh cbq.sh stop"

C.4.2 At the IDS/RealServer compuer (Pluto)

1. Open 3 terminals on Pluto
2. At terminal 1, change directory to "realsystems"
3. Start RealServer

- ./Bin/rmserver rmserver.cfg

4. At terminal 2, change directory to “snort-1.8.6”
5. Start Snort
   - ./snort –A UNSOCK
6. At terminal 3, change directory to “snort-1.8.6/alert”
7. Start the “alert” program
   - ./alert –h titan.uccs.edu
8. To monitor network interface activities, start a new terminal and type “iptraf” at the command prompt
9. To terminate RealServer, Snort, or the “alert” program, type “Clt+C” at their respective terminals
10. To view log files, change directory to /var/log/snort/

C.4.3 At the RealPlayer Client computer (Linux)

1. Open 2 terminals
2. At terminal 1, change directory to “plot-final-0928”
3. To start plotting the number of packets received by network interface eth0 at 1-second interval, execute the command:
   - sh runplot.sh 1 eth0
4. At terminal 2, change directory to “plot-final-0928”

5. Start the graphing utility “gnuplot”:

   - gnuplot input.sh

6. To monitor network interface activities, start a new terminal and type “iptraf” at the command prompt

7. To terminate the plotting programs “runplot.sh” and “gnuplot”, type “Clt+C” at their respective terminals

C.4.4 At the Attack Client “Saturn”

1. Change directory to “Stacheldrahtv4”

2. Verify that the process “mserv” is active using “ps –e”. If the “mserv” process does not exsit, start mserv:

   - ./mserv

3. Change directory to “Stacheldrahtv4/telnec”

4. Start the attack management program “Client”:

   - ./Client saturn.uccs.edu

5. Type in the password when prompted: manager

6. Launch an attack

   - .msyn titan.uccs.edu

7. Terminate the attack

   - .mstop all

8. Exit Stacheldraht

    - .quit