LOAD BALANCING FOR HIGH-SPEED PARALLEL NETWORK INTRUSION DETECTION

A Thesis

Submitted to the Graduate School
of the University of Notre Dame
in Partial Fulfillment of the Requirements
for the Degree of

Master of Science

by

Kyle Bruce Wheeler, B.S.

_____

Lambert Schaelicke, Director

Graduate Program in Computer Science and Engineering
Notre Dame, Indiana
April 2005

LOAD BALANCING FOR HIGH-SPEED PARALLEL NETWORK INTRUSION DETECTION

Abstract

by

Kyle Bruce Wheeler

Network intrusion detection systems (NIDS) are deployed near network gateways to analyze all traffic entering or leaving the network. The traffic at such locations is frequently transmitted in such volumes and speeds that a commodity computer quickly becomes overwhelmed. NIDS must be able to handle all of the traffic available. The SPANIDS platform addresses this problem with a custom hardware load balancer that spreads traffic over several NIDS sensors. The load balancer ensures that sensors do not become overloaded by shifting traffic between sensors while maintaining network flow continuity when possible. The balancer must be resistant to attacks designed to overwhelm it. This work outlines the design of the SPANIDS load balancer and evaluates its performance using simulation. Several design points are examined, including overload detection, locating overload causes, and several overload avoidance techniques. The simulation results confirm the viability of the SPANIDS architecture for scalable parallel network intrusion detection.

To Mom and Dad.

CONTENTS

FIGURES

ACKNOWLEDGMENTS

I would like to acknowledge all of my friends and family, without whose constant support this paper would not be possible. In particular, Emily Kaulbach provided extensive mental support and editing suggestions; Branden Moore provided technical and mathematical assistance and just the right amount of ridicule; and Richard Murphy provided a sounding board for hair-brained ideas. I would also like to thank my advisor, Dr. Lambert Schaelicke, and Curt Freeland, both of whose judgment, guidance, and inspiration was invaluable.

Finally, I would like to thank the U.S. Government and the National Science Foundation for their generous grant, number 0231535, which allowed me to pursue my work.

CHAPTER 1

INTRODUCTION

1.1 Overview

Network intrusion detection systems (NIDS) are generally deployed at choke-points in networks in order to analyze all outgoing and incoming traffic. Unfortunately, the traffic at such locations is frequently transmitted in such large volumes and at such high speeds that a commodity computer cannot receive and analyze all of the traffic. An intrusion detection system's effectiveness is directly related to its ability to analyze all of the traffic it is given — it cannot report what it cannot detect. Network intrusion detection presents a unique network capacity problem because NIDS are typically deployed as transparent systems to prevent attackers from detecting and attacking the NIDS directly. Such transparency requires that the NIDS cannot be in the critical path of network communication and thus cannot request that senders slow down to allow it to process all of the traffic, whereas regular network-connected hosts can. The Scalable Parallel Network Intrusion Detection System (SPANIDS) addresses this problem with a custom hardware load balancer that spreads network traffic over a set of NIDS sensors. The load balancer attempts to ensure that none of the sensors become overloaded while maintaining the continuity of network connection flows when possible so as not to preclude state-

ful traffic analysis. The sensors participate in overload avoidance by notifying the load balancer when their individual packet load is approaching overload levels.

1.2 Background

In terms of efficiency and cost effectiveness, the importance of computer security from exploitation attempts and intrusions via the Internet is increasing rapidly. A complete network security system has many aspects, including parts designed to prevent an attack, parts designed to thwart an ongoing attack, and parts designed to determine whether an attack is occurring. A network intrusion detection system (NIDS) falls into this latter category. Specifically, a NIDS is a system for recognizing network-based attacks by analyzing network traffic. NIDS identify network traffic as one of two categories, "malicious" traffic and "benign" traffic, thus assisting administrators' responsive and defensive efforts. As such, NIDS have become a critical part of secure network-connected systems.

1.2.1 Undetected Attacks are Successful Attacks

NIDS allow administrators to monitor the network and discover when and specifically what malicious activity is occurring. Assuming that malicious activity is equally likely in all traffic, the probability of detecting malicious activity is directly proportional to the percentage of the available traffic that is examined. If a NIDS cannot examine all available traffic, there is a risk that malicious activity will go undetected — a risk that increases as the amount of traffic the NIDS cannot examine increases. Undetected malicious activity

cannot be countered, and is more likely to cause serious damage.

1.2.2 Network Characteristics

Network traffic frequently exhibits "bursty" behavior, meaning that network packets tend to travel in clusters [4]. The packets in clusters typically have very small inter-packet gaps, while at other times network packets are more spread out with larger inter-packet gaps. An inter-packet gap is the time between the end of one packet and the beginning of the next packet. When bursts of packets arrive at a network-connected host, the packets may arrive faster than that host can receive them. To compensate for this inability to handle temporary bursts of rapidly arriving network packets, most computer systems use buffers to store packets until they can be processed. Once the buffers are filled, further packets cannot be stored for later processing and will be ignored, or "dropped."

In ordinary TCP network traffic, the sender and receiver cooperate to adapt their rate of communication to their ability to communicate and to the load the network will bear. For instance, if the sender sends packets faster than the receiver can understand them, the receiver will not acknowledge them. Missing acknowledgments will cause the sender to reduce the number of packets it may send while awaiting acknowledgments, thus reducing communication speed. Unlike such typical network communication, a NIDS is not in the critical path of network traffic: it is neither the sender nor the primary receiver of the traffic. Most NIDS attempt to be undetectable, eavesdropping on network communications rather than participating in them. Some systems build upon NIDS for automated response, but a pure NIDS is just a sensor. Because

a NIDS cannot participate in the communications it is monitoring, it cannot affect the rate of that communication. Therefore it must be able to analyze packets at least as quickly as any host on the protected network in order to be effective. A NIDS generally tracks multiple network conversations at once, requiring it to be even faster still.

1.2.3 Hardware Characteristics

In a typical network-connected computer, when a packet arrives via the network, the packet is first received by the network interface controller (NIC), which decides whether to notify the computer's operating system of the packet's arrival based on the packet's destination. For example, in Ethernet networks, NICs use the MAC address to filter packets. However, since NIDS listen to all network traffic, the NIC cannot be allowed to shield the operating system from traffic not addressed to it. Thus, in a computer performing network intrusion detection, all packets cause the NIC to notify the operating system. The NIC notifies the operating system by generating a hardware interrupt and copies the contents of the packet into main memory. The interrupt, the resulting system call, the context switch because of the interrupt, and moving the packet into the operating system's packet buffer takes time away from the packet analysis work the network intrusion detection sensor software must do. More importantly, in the time it takes the computer to enqueue the new packet and return to analyzing previously received packets, another packet may arrive that needs to be enqueued. Such a quick turn-around prevents the computer from making progress in analyzing received packets and emptying the packet buffer. New packet arrivals preventing analysis of already buffered

packets is particularly problematic in systems on very fast network connections with small inter-packet gaps.

1.2.4 Basic Network Intrusion Detection

The most basic NIDS layout is a single system connected to a shared network of some kind — Ethernet is popular [4] — listening to all of the traffic on the network. Software on this computer takes every packet it receives from the network and analyzes it, looking for malicious traffic. An example of this layout is illustrated in Figure 1.1.

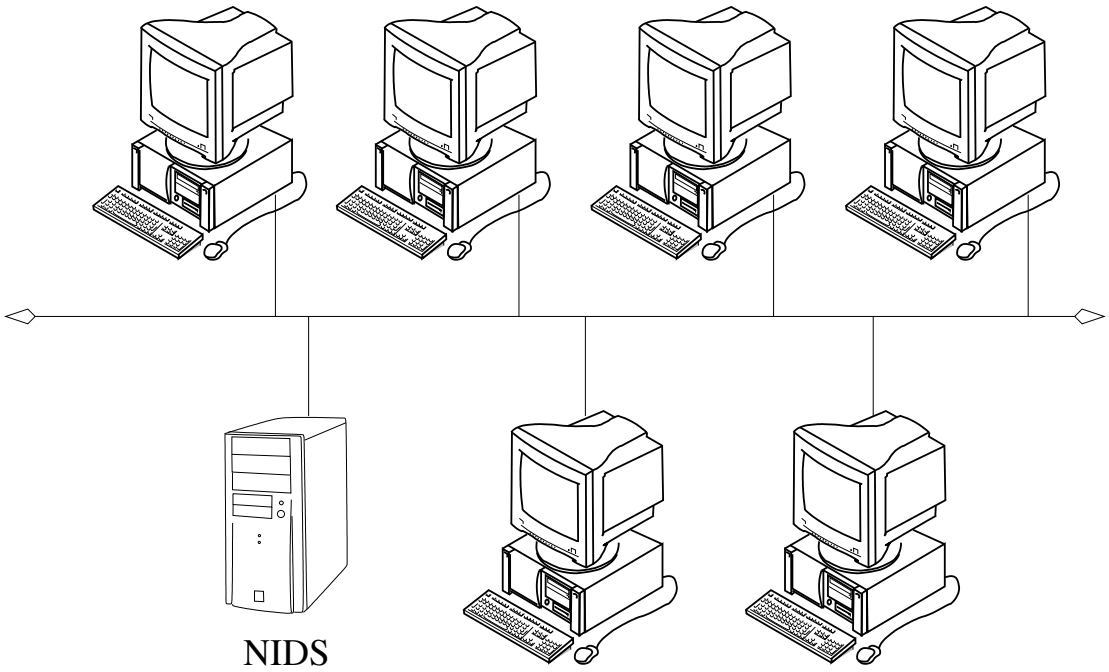


Figure 1.1. Shared Network NIDS Layout

Shared, or hub-based Ethernet networks, however, are being replaced by switch-based Ethernet networks. On carrier-detect networks like Ethernet, switch-based networks protect network connected hosts from the unnecessary interference of other hosts on the network, thus increasing the average available bandwidth by reducing the scope of a packet transmission. In so doing, switch-based networks prevent hosts on the network from seeing all traffic on the network, prompting a move of NIDS from the middle of the network to choke-points. A choke-point in a network is generally the gateway from the network to other networks and the Internet. Companies and universities that have large networks generally pay for a single connection to the Internet and perhaps a second fail-over connection, both of which are choke-points. An example of this layout is illustrated in Figure 1.2. While a NIDS at a choke-point cannot receive all of the traffic on the protected network, it can receive all traffic entering or leaving the protected network. Thus, at this one point, one can monitor both incoming attacks and out-bound infections. Notice that the NIDS is still not designed to participate in the critical path of communication, and in fact receives a copy of the network traffic sent to the router from a one-way network tap.

There are some security systems that use NIDS to interfere with the network [19], but most do not. Some networks have NIDS scattered throughout the network to obtain more detailed information about possible malicious activity within the network. However, the most common location for a single NIDS deployment is at a choke-point.

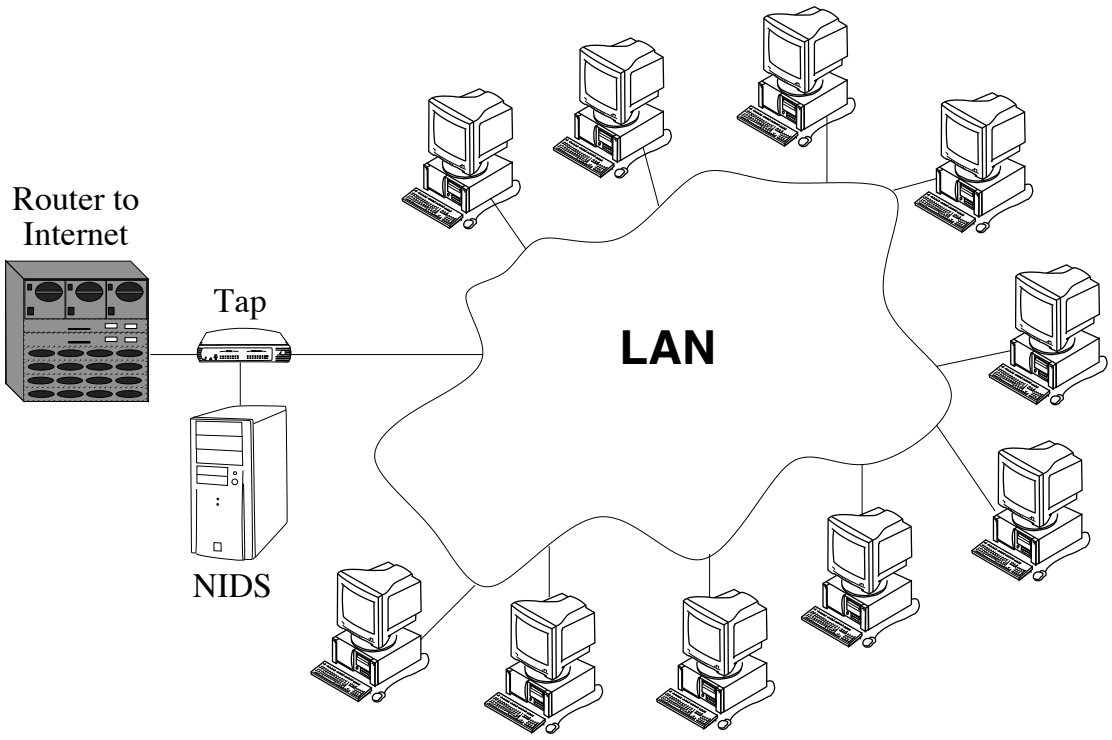


Figure 1.2. Choke-point NIDS Network Layout

1.3 The Problem

1.3.1 Networks Are Too Fast

Network traffic speeds and volume are increasing at an exponential rate. Simple Ethernet networks have increased in speed tenfold almost every four years over the past few decades. Bandwidth has accelerated from ten megabits per second 10Base-T Ethernet in 1991, to 100 megabits per second 100Base-T Ethernet in 1995, to one gigabit per second Ethernet in 1998, and most recently to ten gigabits per second Ethernet in 2002. This increase in speed far outstrips the ability of any single network host to keep track of every packet

transmitted on a network [8, 32]. For instance, Schaelicke et al used the industry standard NIDS software Snort on several different processors each equipped with a 100 megabits per second Ethernet card. The most capable system tested, a 1.2GHz Pentium III, dropped significant numbers of packets with a mere 560 attack signatures to match against each packet — far shy of the default set of 2,910 attack signatures that come with Snort. In Schaelicke's experiments, the Pentium III did better than even faster Pentium IV systems. As Pentium IV CPUs have larger pipelines and thus more interrupt overhead, this indicates that the amount of interrupt overhead has a significant impact on the ability of a processor to analyze packets. The gain in clock speed between the Pentium III and Pentium IV systems was not sufficient to overcome the effect of the additional interrupt overhead from the Pentium IV's deeper pipeline; thus packet buffers filled with unprocessed packets and packets were lost [32].

Schaelicke's experiments demonstrate that the impact of interrupt overhead is a problem that will not be alleviated by simply waiting for faster processors to be developed. The methods that are employed to increase microprocessor speeds are typically related directly to increasing the pipeline depth. Increased pipeline depth allows microprocessors to operate at higher clock speeds, increasing throughput in terms of number of instructions per second. Unfortunately, a larger pipeline requires the storage of more state whenever there is a context switch. A context switch occurs every time there is a hardware interrupt. If more state must be stored in order to handle an interrupt, more time will be required to handle the interrupt [23]. Thus, the increase in interrupt handling overhead mitigates the clock speed gains for

applications that rely heavily on hardware interrupts such as network communication.

In addition to transmitting packets very quickly, next generation networks have a larger volume of packets than older networks. Extremely high bandwidth network connections that need to be monitored can carry orders of magnitude more data than a computer can handle. For example, typical high-end memory bus bandwidth on commodity systems is about 16 gigabits per second, while an OC-3072 transmits data at over 160 gigabits per second [16]. Network connections are generally designed to be used by many computers at once, all trying to communicate at full speed. For this reason, aggregate network bandwidth will likely remain far ahead of any single computer's ability to process it.

1.3.2 Custom Hardware is Inflexible

There are two general methods for addressing the inability of common microprocessor architectures to deal with the overhead associated with network communication and network intrusion detection as the speeds of networks increase. The first frequently used method involves carefully tuning existing microprocessors to operate at the speed of the target network and offloading as much of the network communication overhead as possible to customized hardware. However, there are several problems associated with this method. First, while tailoring microprocessors to the specific speed of the network, timing tricks are employed that are difficult to repeat. Next, designing systems to handle next-generation networks at gigabits per second and faster is extremely difficult and expensive. When the network that needs to

be monitored increases in speed, such custom systems are easily overloaded and must be both redesigned and replaced, incurring additional cost. Additionally, while this method can be effective with current hardware and some slower networks, such techniques provide only a small factor of improvement in traffic handling speed. Networks are increasing in speed more quickly than Moore's Law, and the relatively small factor of improvement this technique provides cannot affect the growth-curve disparity in the long run. Finally, such techniques cannot compensate for any disparity between computer bus bandwidth and network bandwidth.

1.3.3 Offloading Overhead Has Limits

The second method for addressing the problem of interrupt overhead is a technique known as "interrupt coalescing." In a system that uses this technique, the NIC is augmented with more memory and a queuing system. As packets arrive at the NIC they are stored, several packets at a time, in on-chip memory. The operating system is then notified of the arrival of packet clusters rather than individual packets. This technique can change the overhead-per-packet ratio significantly and has the potential to scale better than the previous technique, particularly since latency is not a concern for network intrusion detection. However, this technique does not address CPU processing limitations or pure memory bandwidth limitations and thus cannot cope with extremely fast networks or very large amounts of bandwidth.

1.4 Contribution

Unlike approaches aimed at eliminating overhead, like interrupt coalescing, or approaches targeting memory latency and single-host processing speed, the SPANIDS project addresses the network intrusion detection capacity problem by creating a way that several sensors can cooperate to handle the full network bandwidth even on very fast network links. By cooperating with a small custom load balancer, sensors can reduce the number of dropped packets drastically, even when using commodity sensors. The system is also scalable, as additional capacity can be added to the system by adding more cooperating sensors.

CHAPTER 2

SPANIDS DESIGN

2.1 Intrusion Detection Philosophies

There are several general techniques that NIDS use to categorize traffic into "malicious" and "benign" traffic. For example, the software package Bro uses a philosophy similar to that employed by many firewall administrators: anything that is not expressly permitted must not only be forbidden but malicious [28]. This philosophy provides complete protection against unknown attacks, and rules that are too restrictive tend to be exposed quickly. Because the full range of acceptable traffic is significantly smaller than the full range of potentially malicious traffic, the analysis stage of Bro-like NIDS detectors can be very fast. Unless extremely carefully written, however, the Bro approach can lead to either a very large number of false positives, or large loopholes as a result of a misguided effort to reduce false positives. Also, as the scope of acceptable network use increases, more valid-traffic recognition must be coded into Bro.

The most popular technique is the opposite approach; most traffic is considered benign except for traffic that matches known attack signatures. This approach is used by the open-source software package Snort [13, 31]. Snort uses a long list of known attacks and attack signatures. These signatures are

generally byte patterns that have been observed in network-based attacks by others who use the software and have contributed the byte patterns to the Snort-using community. The primary complexity of the Snort-like approach is the pattern-matching algorithm — such as the Aho-Corasik [1, 27], Boyer-Moore [2], or Wu-Manber [35, 36] algorithms — that are used to search captured packets for known malicious byte patterns.

2.2 Parallel NIDS Requirements

The problem that networks are faster than the computers they serve, and how this disparity makes high-quality network intrusion detection extremely difficult is now well understood. In situations where many computations need to be performed but the outcome of any one computation does not depend on the others, one solution is to perform the computations in parallel. The first step to designing a parallel traffic analysis solution is to establish the basic requirements that are not speed related.

2.2.1 Flows

2.2.1.1 What is a Flow

When a host on a network wishes to send information to another host on the network, the nearly universal mechanism is to segment the information into chunks and then to use either TCP/IP or UDP/IP encapsulation for each of the chunks. Each encapsulated chunk, or "packet," is then sent over the network to the destination host, which reassembles the packets into the original data. Nearly all network traffic consists of these "connections" or "flows" of related packets. In the case of TCP, a flow simply corresponds to a TCP con-

nection, but even datagram protocols such as UDP often show connection-like behavior. For instance, each NFS client maintains a logical connection to the server, even though the underlying communication protocol is based on datagrams and is inherently connection-less. These flows can be roughly identified based on the tuple of source and destination host IP addresses and source and destination TCP or UDP port numbers. Generally, two hosts communicating on the network will distinguish unrelated flows of packets between them by using different port numbers of each unrelated flow. However, since there are only 65,535 distinct ports, port numbers will be reused. Thus, for correct flow disambiguation, it is critical to consider both time and the state of the flow. In the former case, if a flow has gone idle for a sufficient time period, the flow can usually be considered ended and further packets with the same address/port tuple can be considered a new flow. In the latter case, a flow may be ended by either side, at which point the address/port tuple may be reused at any time.

2.2.1.2 Flow Clustering

In order for a NIDS to work best, it must be able to determine what data any given host will receive from the network. Many network-based attacks can be encapsulated in a single packet — some cannot work unless transmitted in a single packet. However, there are a large number of network-based attacks that do not have this restriction and can easily be split across several small packets. Each single packet may contain as little as a single byte of data, which by itself is insufficient to consider malicious. But when the destination host reassembles all of the packets in the flow, they may form a malicious

byte pattern. By dividing the attack into several small packets, an attacker can evade detection by a NIDS capable solely of individual packet analysis.

Stateful flow-based traffic analysis requires that a sensor observes the complete set of packets in a flow. This allows the sensor to assess the state transitions of the traffic and reassemble the segments of data in order to determine precisely what data each destination host will receive. A high-quality distribution mechanism will not inhibit stateful analysis by dividing network flows between several parallel NIDS sensors. Avoiding breaking network flows across multiple sensors requires some ability to identify and track flows so that individual packets associated with a flow all will be sent to the same sensor as other packets in the flow.

2.2.1.3 Flow Clustering Challenges

Unfortunately, fully distinguishing separate network flows — TCP, UDP, and others — is prohibitively expensive in terms of storage. The volume of possible flows needing to be tracked in a single second on an extremely fast network is overwhelming. TCP flows, for example, have a maximum timeout of approximately five minutes. Additionally, only a single 64 byte TCP SYN packet is required to start a TCP flow over Ethernet (over other communication mediums, the minimum size may be as low as 40 bytes). In order for a system tracking individual flows to be impervious to a trivial resource-exhaustion attack, the system needs to be able to handle the extreme case of every packet over a five-minute time span being a SYN packet. On a communication medium like Gigabit Ethernet there can be 1,488,100 packets per second [14]. That many SYN packets every second, over the course of the five

minute TCP connection timeout, creates 446,430,000 new TCP connections that must be tracked. Even if tracking any single flow is cheap, when every individual flow is tracked, a simple resource exhaustion attack will overload all but a perfect system.

If packets from broken connections can be logged by the sensors in a sufficiently rapid manner, it may be possible to do some analysis on the logged packets. If this can be done, it is possible to correct the problem of splitting flows by analyzing all of the packets logged by all of the sensors and doing detailed flow-analysis of the entire collection of packets. Unfortunately, this is not a general solution to the problem of flow tracking. If large quantities of packets are logged and subjected to such detailed analysis, eventually the log analyzer will fall behind. If analysis falls behind the arrival of new data, eventually there will come a point when either the storage for logging packets is full and further packets will be dropped or the detailed analysis is so far behind that any evidence of maliciousness discovered is too old to be useful.

2.2.1.4 Flow Requirements

As discussed above, a good distribution mechanism should avoid distributing packets belonging to a single flow over multiple sensors. Additionally, a secure distribution mechanism must be able to withstand attacks that target its method of identifying flows, such as resource exhaustion attacks.

2.2.2 Network Parallelism

2.2.2.1 Packet-level Parallelism

Network traffic is inherently parallel on several levels. At the lowest logical level, the network is a mass of individual packets, any of which can be a threat. A simple technique for exploiting this level of parallelism is known as round-robin distribution, which distributes an equal number of packets to all sensors. This technique involves rotating the sensor assigned to each sequential packet around the set of available sensors. Such an approach evenly distributes network traffic along a useful logical fault line to achieve parallelism. However, this technique does not preserve flows, and is thus subject to the possible blind spots outlined in Section 2.2.1.2.

2.2.2.2 Flow-level Parallelism

Packet-level parallelism is not a very good level of parallelism to exploit because keeping flows together is desirable. Thus, exploiting the parallelism of the flows of network traffic is more useful. Traffic transmitted through choke-points where NIDS are typically deployed is usually an aggregate of large numbers of unrelated small network flows. Nevil Brownlee and K. C. Claffy, working on the NeTraMet project, have found that approximately 75% of TCP network traffic is less than 10 packets or two kilobytes in length, and 90% is less than 20 packets or 20 kilobytes in length [3]. This means that the usual bandwidth that NIDS must analyze normally consists of very large numbers of small flows, each of which can be analyzed independently.

2.2.2.3 Flow-level Parallelism Problems

Relying solely on flows to determine how to distribute network traffic across the sensors in a parallel NIDS has a major drawback. Flows are controlled by the hosts using the network. The general characteristics of network traffic flows — such as average packet size, inter-arrival times, and the balance of protocols used — change regularly due to the changing needs of the hosts on the network over time. For instance, an ISP may experience mostly HTTP traffic during the day, followed by large UDP packets used by file sharing programs in the evenings. Such shifts in network characteristics may cause shifts in the load distributed across the set of parallel sensors, possibly overloading some of the sensors. There is also the possibility that an attacker may maliciously manipulate network traffic characteristics in an attempt to avoid NIDS scrutiny. As an extreme example, an attacker could ensure that all packets in the network appear to be part of the same flow. If each flow is always directed to a single sensor by the distribution mechanism, the single sensor to receive the massive flow would be instantly overloaded and the other sensors in the parallel NIDS would receive no traffic, rendering them essentially useless. However, circumstances need not be so extreme to cause problems; traffic characteristics need only shift enough to cause the distribution mechanism to send more packets to a single sensor than it can handle in order to lose packets and thereby allow some traffic to go unexamined. Succinctly, even a single flow can overload a sensor. Thus, a strict distribution of entire flows to sensors may still result in packet loss.

2.2.2.4 Parallelism Requirements

To tolerate shifting non-uniform network characteristics and high-bandwidth flows, packet loss avoidance requires the distribution mechanism to be aware of excess load on any one sensor node and to be able to adjust the distribution of network traffic across the sensors. In the extreme case of a single network flow consisting of sufficient packets to overload a sensor, it may be necessary to fall back to a packet-level distribution scheme that ignores flows, at least for a sufficient amount of network traffic to alleviate the problem. Such an adaptive approach might expose the system to flow-based segmentation attacks, but it is far better than dropping packets and allowing them to go unexamined.

Packet-level distribution in the case of flows capable of overloading a single sensor is not as unfortunate as it is in the general case. Presuming that the sensors can analyze packets at least a quickly as any single host in the protected network can receive packets, a flow that overloads a sensor is a flow that will overload its target host as well. Thus, if the goal of the attack is more complicated than a simple overload attack, the attack packets must contain sufficient malicious data to be effective even if the target host does not receive all of them. In such a circumstance, a packet-level distribution scheme will likely not inhibit detection of the malicious payload.

2.2.3 Worst Case Planning

A parallel NIDS mechanism should not introduce any additional vulnerabilities to the NIDS. As discussed, existing single-host NIDS can be overloaded with a flood of small packets, similar to a conventional denial-of-service at-

tack. While a parallel NIDS directly addresses the capacity problem, it may introduce new vulnerabilities through the distribution mechanism and load balancing algorithm. The distribution mechanism must be able to handle and recover from worst-case scenarios.

2.3 Layout

A generic NIDS, as illustrated in Figure 2.1, can be expanded to a generic parallel NIDS with the introduction of a "splitter" module, as illustrated in Figure 2.2. The splitter module is some set of hardware that will divide the network traffic in some way and send those divisions to several NIDS sensors. There are two basic approaches to organizing and implementing the splitter section of the generic architecture illustrated in Figure 2.2. The first is to make the splitter a part of the intrusion detection analysis framework, and the second is to make the splitter a generic network flow separator.

2.3.1 Framework-based Splitting

Splitting the network traffic up in an intelligent manner that is a part of the analysis and attack-detection functionality of the overall NIDS can be an effective way to limit the amount of traffic to any single sensor.

Sensors generally have two primary operations: first to figure out what to look for in a given packet, and then to look for it. The first of these two operations is frequently done by using simple rules based on the packet headers to determine what content-based rules apply to the packet. Once the packet category has been determined, the packet is searched for any of the many known malicious byte patterns or "attack signatures" that apply to that cate-

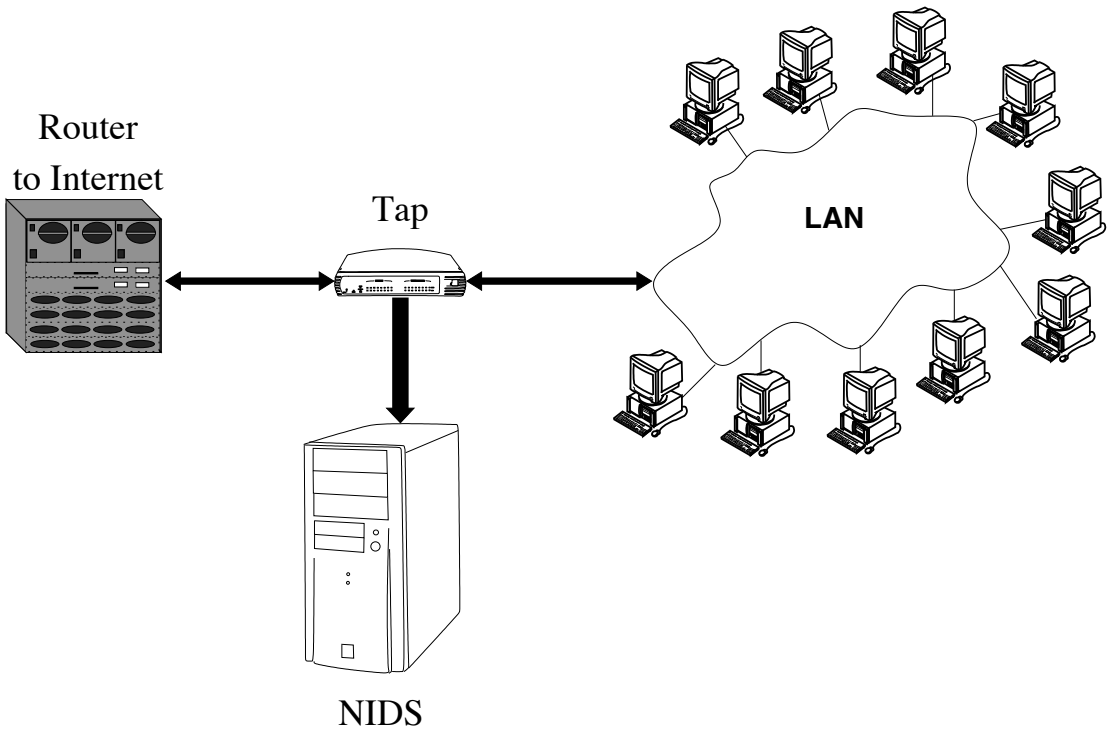


Figure 2.1. Generic NIDS

gory. It is frequently the case that single, simple rules allow the sensor to categorize packets very broadly and very quickly. In so doing, such simple rules reduce the number of attack signatures that must be searched for in captured network traffic. Similarly, it would be possible to separate this categorization from the sensor itself to a separate network splitter, so as to quickly divide the network traffic into known categories, each of which is directed to a dedicated sensor for that category. This separation, generally, would drastically reduce the amount of network traffic and thus number of packets that any one sensor would need to examine. Additionally, such separation would reduce the number of patterns that any one sensor must search for in the traffic it re-

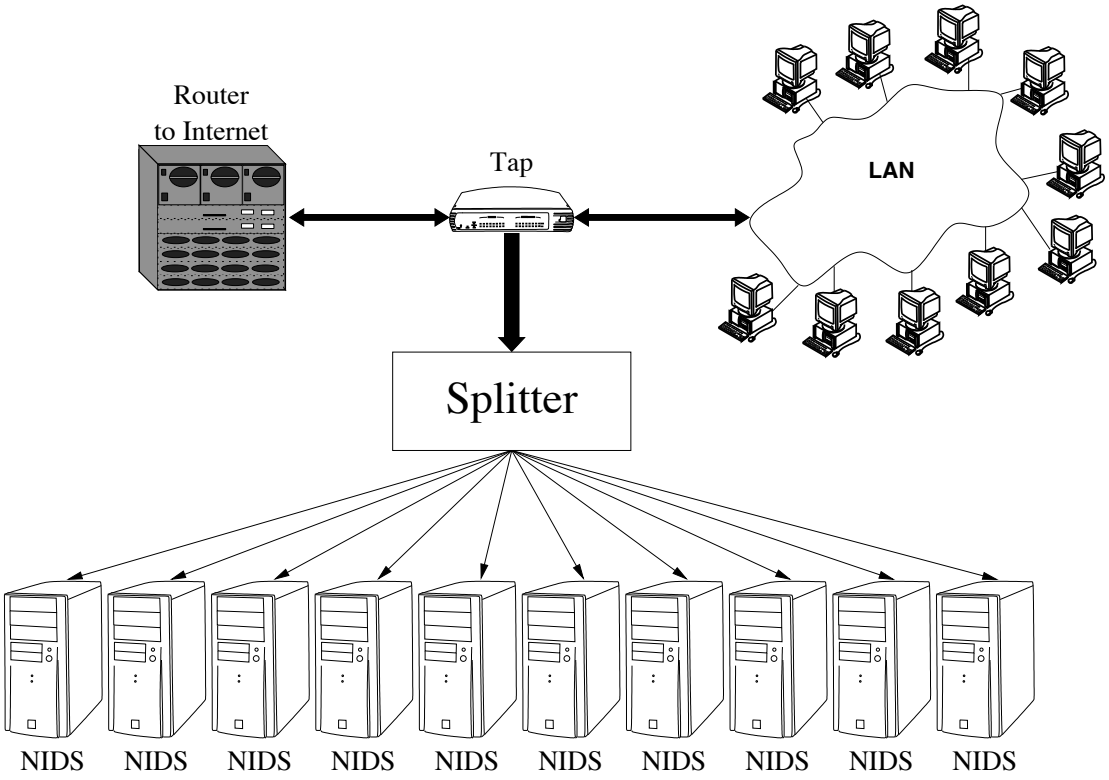


Figure 2.2. Generic Parallel NIDS

ceives, improving the sensor software's efficiency and thus capacity to handle large amounts of traffic without negatively impacting accuracy.

For example, as illustrated in Figure 2.3, if the only traffic that the sensors know how to analyze is HTTP, FTP, and SMTP traffic, then it is easy for a splitter to simply send all HTTP packets one way, all FTP packets another way, all SMTP packets a third way, and either drop everything else or send it to a general-purpose sensor. This initial splitter can send the packets to sensors or even to more detailed splitters. Subsequent splitters may be able to do even more logical separation of the network traffic. The more hierarchical logical

separations occur, the less traffic travels any one route through the system, and the sensors that do the analysis get a very small slice of the original network traffic sent to the NIDS.

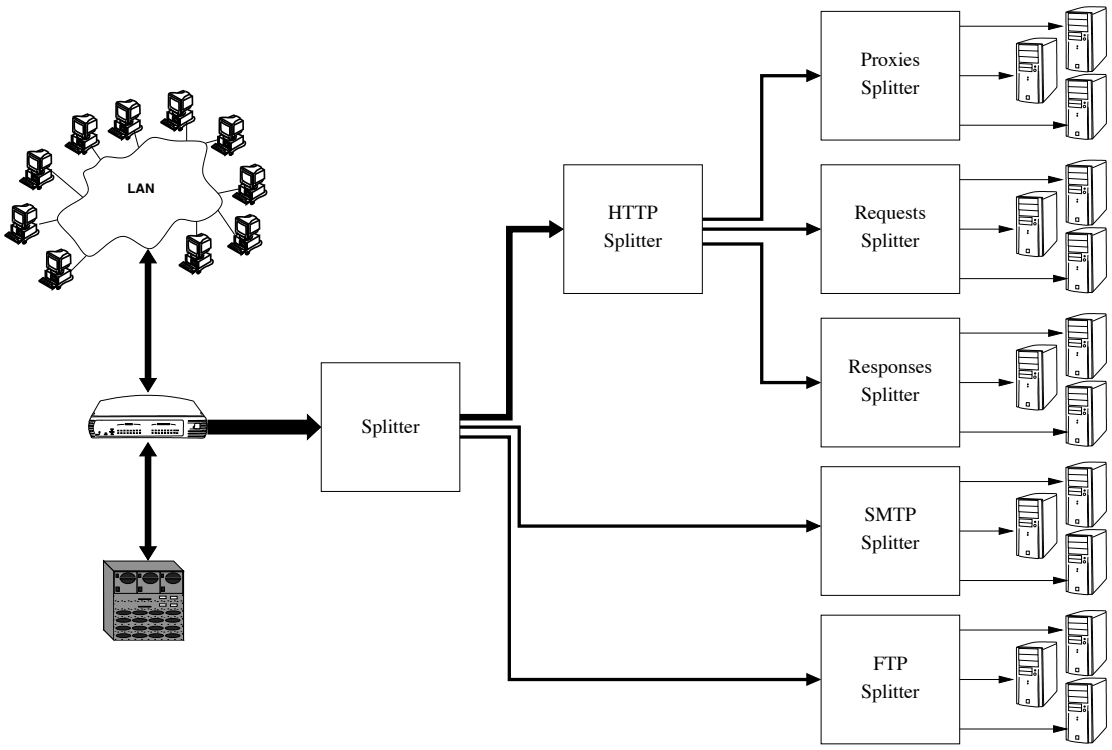


Figure 2.3. Simple Framework-based Splitting Example

There are some critical drawbacks to this approach to distributing the network traffic across an array of sensors. One drawback is that the hierarchical layout is not easily reconfigurable. If new attack vectors are discovered and new rules for the sensors are written, the layout of the splitters may need to

be re-organized, which will likely require that the whole NIDS be taken offline long enough to reconfigure the splitter-tree. Another problem with including too much of the analysis system in the splitters is that the tree is not detector-agnostic, which is to say, the splitter tree is tied to a particular network traffic analysis software package, and more generally to a particular network traffic analysis schema. Thus, changing or adding network traffic analysis software may require reconfiguring the splitter tree, or may even be impossible if the new analysis software does not lend itself to such offloading very well.

The primary, and most critical drawback of distributing traffic in this manner is the problem of variable network characteristics, the extreme form of which are narrow denial of service attacks (NDoS). NDoS attacks are denial of service attacks whose packets are very similar or closely associated in the logic of the detection system. Using parts of the detection logic to divide traffic among several systems to avoid overloading them makes it difficult if not impossible for the overall system to detect and handle load changes due to variations in traffic characteristics. Thus, such a system cannot respond effectively to a sudden dramatic increase of traffic that follows a specific path through the splitter tree. Without some adaptivity during an NDoS attack, a single or a group of sensors may be required to suddenly deal with as much as the full network bandwidth. In generalized non-hostile traffic, this would likely not be a problem. Nevertheless, traffic mixes may change and require more or less of the analysis logic to be offloaded into the splitter tree.

2.3.2 Generic Splitting

The other method of splitting network traffic, and the one used by SPANIDS, is to divide the traffic according to much simpler and somewhat more arbitrary criteria. Such a splitter must, rather than splitting along higher-level protocol boundaries, split traffic up according to an inherent level of parallelism within the traffic, as discussed in Section 2.2.2. The approach used in the SPANIDS system is to split traffic along flow boundaries whenever possible, and packet boundaries when flow-based splitting is insufficient. Network packets belonging to the same flow or connection are uniquely identified by the tuple consisting of source and destination IP addresses and port numbers. To distribute packets over a set of sensors, the splitter generates a hash value for each packet based on the fields in that tuple. Each hash value is associated with one of the sensor nodes the splitter is dividing the traffic between, and dictates which sensor will receive the packet.

This approach addresses several of the requirements outlined in Section 2.2.1.2. It does not limit the maximum number of concurrent flows or connections, since the hash table size is fixed and independent of the number of flows. At the same time, the design is sufficiently flexible to support varying numbers of sensors with a single splitter device. Most importantly, network packets belonging to the same flow always hash to the same value and are thus always forwarded to the same sensor node without incurring the cost of keeping track of individual flows. In addition, this generic hash-based approach to traffic splitting allows for greater flexibility in the system to adapt to changes in network characteristics.

2.4 Prototype Hardware

Though it is not part of the work presented here, it is useful to note that a prototype implementation of the scalable distributed NIDS architecture of SPANIDS is being implemented that will eventually provide more realistic data to corroborate the simulation output presented in the next chapter. This effort ensures that the design decisions described in this paper result in a truly scalable system capable of processing packets at wire speed, and also facilitates extensive performance characterizations under realistic network conditions. The prototype system consists of a Xilinx Virtex-II FPGA [5] hosted in the PCI slot of a commodity computer running the Linux operating system, a commodity switch that implements the data plane of the load balancer, and a collection of rack-mount systems as sensor nodes.

For each packet, the control logic determines the destination sensor and rewrites the destination MAC address accordingly. The external switch then forwards the packet to the associated sensor. Splitting the actual load balancer into the FPGA-based control logic and a data plane reduces the number of network ports required on the FPGA board, and also reduces the logic design effort. Sensor hosts are running the open-source Snort network intrusion detection software [13, 31], though there is no impediment to installing different network intrusion detection software on them. A custom kernel module on each sensor implements a POSIX RAW packet interface [15] that monitors its buffer use and issues flow control messages to the load balancer as necessary. A second switch routes alert messages from the sensor nodes to a database system for further analysis and long-term storage.

2.5 The Simulation

The SPANIDS system, like many hardware systems, is more convenient to simulate than to build. To evaluate the performance of the SPANIDS load balancer, and to further refine the design and explore tradeoffs, a trace-based event-driven simulator has been developed. Trace-based simulation is the ideal tool for this purpose, as network traces can provide realistic workloads to the simulator without needing the simulator to go through the overhead of fully simulating traffic sources. This setup also closely corresponds to a real implementation where a NIDS platform is offered a certain traffic load without being able to influence it. The simulator implements a detailed and functionally accurate model of the load balancer operation, including the hash functions and tables, responses to flow control messages and promotion/demotion of hash buckets.

A configurable number of sensor nodes are modeled as finite packet buffers that drain packets at a rate based on the packet size. When the buffer use reaches a configurable threshold, a flow control message is issued to the load balancer. In the next chapter this simulator will be used to evaluate trends and relative success of several design considerations.

Note that the simulator does not model the actual intrusion detection software, as these details have no impact on the load balancer and would merely provide more accurate packet drain speeds. Instead, the main focus of the tool is to characterize the load balancer behavior and to refine design decisions such as suitable traffic intensity measures, time-out values, and hash table sizes.

2.5.1 Trace Files

Two different network traces, both captured on the University of Notre Dame campus Internet connection, are used for the simulations presented in the next chapter. An older trace, which will be referred to as the "slow" trace in this thesis, was recorded from a 45 megabits per second link. The slow trace contains 1,402,229,593 packets captured over a 21 hour period on December 1, 2002. The second trace, which will be called the "fast" trace in this thesis, was recorded from a 200 megabits per second link. The fast trace contains 38,599,994 packets captured over a forty minute period on June 9, 2004. These two traces do not contain denial of service attacks, though it is probable that they contain several other types of malicious traffic.

2.5.2 Timing Accuracy

It should be noted that the traces described above are standard tcpdump [17] traces. Tcpdump traces record timestamps for every packet indicating its arrival time with a resolution of microseconds. Even for a network as slow as ten megabit per second Ethernet, microseconds are insufficient to fully model arrival times. Ethernet network links generally have an inter-frame gap (IFG), also known as the inter-packet gap (IPG), of 96 bit-times. A bit-time is the amount of time necessary to transmit a single bit. Ten megabits per second Ethernet has a minimum IFG of 9.6 microseconds, 100 megabits per second Ethernet has a minimum IFG of 0.96 microseconds, and one gigabit per second Ethernet can have an even smaller IFG that depends on the medium — in these simulations it is estimated to be 0.144 microseconds. Due to the inherent inaccuracy of the timestamps stored in the trace files, many packets have

identical timestamps. It is therefore sometimes necessary to enforce minimum IFGs when the traces are used in simulation. In aggregate, the trends should be accurate. Because trace files cannot contain sufficiently accurate data, however, the simulator cannot exactly duplicate real world behavior.

2.5.3 Default Settings

In the simulations, certain defaults had to be chosen to enable comparison between radically different design considerations and to maintain a consistent point of reference. In the next chapter the options available in each critical section will be examined and evaluated based on simulation results. Many of the simulator's settings correspond to design alternatives that will be described and discussed in detail in the next chapter. Nevertheless the defaults are listed here for reference.

For the simulations in the next chapter the "default" settings are as follows, and are used unless otherwise stated. The simulation will load balance between twelve identical sensors. Each sensor is equipped with a Linux default buffer size of 64 kilobytes, and uses estimated overhead that approximates a 1.2GHz Pentium III processor. The simulated sensors have an interrupt and system call overhead of 14.5 microseconds, a per-byte copy overhead of 0.931 nanoseconds (approximately 1GB/second memory bus bandwidth), a per-header-rule analysis overhead of 68.54 nanoseconds, a per-byte-per-body-rule analysis overhead of 0.1356 nanoseconds, and a default set of 600 header rules and 2,528 payload rules. The sensors generate feedback a maximum of 100 times per second, and only generate feedback if their 64 kilobyte buffer is more than 30 percent full. The network medium is assumed to be one

gigabit per second Ethernet with an inter-frame gap of 0.144 microseconds. Other load balancer decisions, such as when to move or promote a packet and where to move a packet, use a Mersenne twister random function [25] to choose between equally probable alternatives. The random function is initialized with the prime number 7177 in all simulations. The default method of locating hot spots is to use byte counts. Finally, when a feedback packet is received, a default of four buckets are affected.

2.5.4 Broken Connections Estimate

As previously discussed, every connection has a distinguishing tuple: its source and destination IP addresses and TCP/UDP port numbers. When connections end, that tuple may be reused to designate a new connection. To estimate the number of connection in the trace files, each tuple is recorded. A TCP FIN packet designates the end of a connection. Thus, when such packets are encountered, further packets with the same tuple can be considered a new connection. The number of tuples in the trace files, with the modification for FIN packets, can be used as a rough estimation of the number of connections in the trace. This estimation technique can only provide a rough estimate as it does not take connection timeouts, retransmission, or advanced flow state into account.

Using this rough estimation technique, analysis of the two trace files indicates that there are 4,493,196 estimated connections in the fast trace and 101,762,453 estimated connections in the slow trace. Extending this technique, each tracked connection is associated with a sensor that it has been sent to. When a packet is routed to a different sensor than the one recorded

for that packet's tuple, the tuple is changed to reflect the new sensor assignment and the connection is considered to have been broken.

CHAPTER 3

SYSTEM DETAILS

Now that the general details of the problem's overall structure and a general idea of the approach that SPANIDS takes has been established, the specifics can be examined.

3.1 The Big Picture

In order to understand the design decisions that have been made and could have been made, one first must understand the overall design and how it fits together. Conceptually, SPANIDS looks something like Figure 3.1. The load balancer unit is where a sensor is chosen for each incoming packet. The load balancer puts the hardware MAC address of the chosen sensor into the destination MAC address field of the packet's Ethernet frame and sends it to the switch. The switch routes the packet to the sensor specified by the destination MAC address.

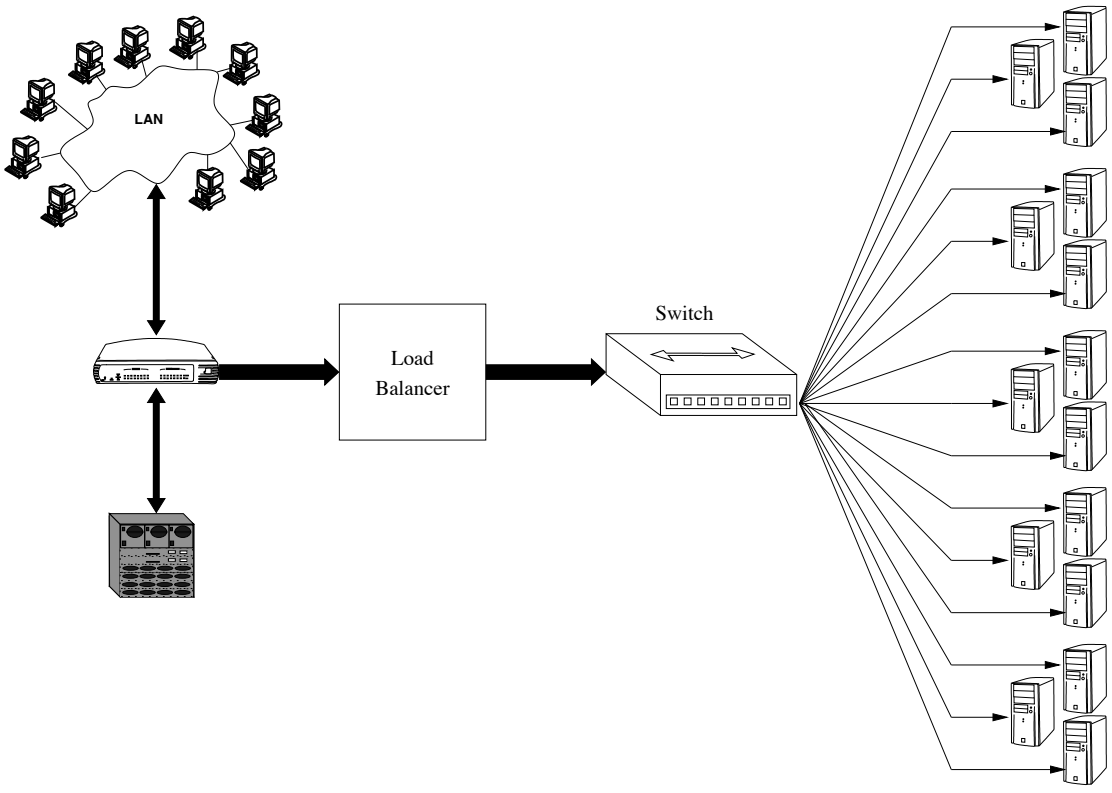


Figure 3.1. The High-Level Architecture of SPANIDS

When a packet is received by the load balancer, a hash of the packet's header information is calculated. This hash is used as an index into a routing table. Each entry in the routing table contains a sensor identifier that is used to tag the packet before emitting it to the switch as described above.

In the sensors, the NIDS software uses a buffered POSIX RAW socket to receive packets from the network. The socket is modified to monitor its packet buffer. The modified socket generates "feedback" packets whenever it determines that the sensor is being overloaded by the network traffic it is receiving.

The feedback packets are transmitted to the switch, which routes them to the load balancer. In the load balancer, receipt of a feedback packet indicates that one of the sensors is being overloaded and the routing table is modified to route traffic away from the overloaded sensor.

The first critical area of this design is the hashing mechanism that is the basis for the routing decisions. A hash must be designed that can operate at line speed, won't separate flows, and can extract sufficient entropy from the incoming traffic to be useful in scattering traffic across the available sensors. The second critical area of this design is the determination of the cause of overload. To respond to feedback, the load balancer must locate the traffic that is responsible for the overload. The third critical area of the design is the routing table, which is inextricably linked to the load balancer's capacity to alleviate overload. This design point must address two issues: what the load balancer should do to mitigate impending sensor overload and the design of the routing decision framework to allow the load balancer to respond effectively to feedback packets. The fourth critical area of the design is the sensors' self monitoring. Sensors must accurately predict that they will drop packets with enough lead-time that the load balancer can avert or mitigate the overload with a minimum of packets lost.

In other words, the critical areas can be summed up as: how should traffic be divided up, if there is a problem what traffic is causing the problem, how should the problem be resolved, and how does a sensor know if there is a problem?

3.2 Hashing

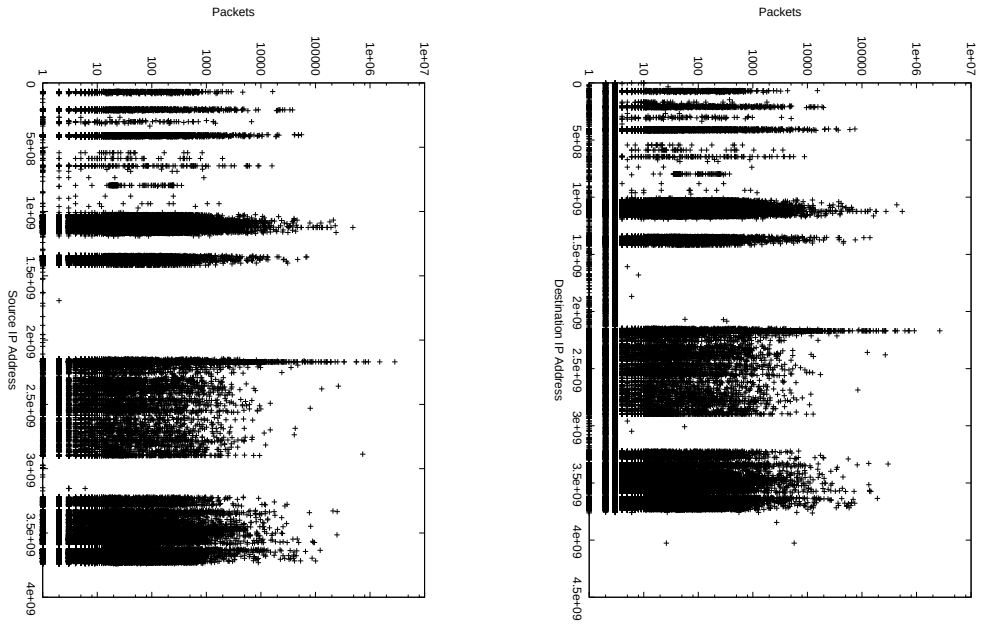
Splitting the network traffic up into chunks in an unintelligent way is easy: split based on the length of the packet, or a count of the number of 1-bits in the packet. However, a good parallel NIDS requires a more intelligent hashing system.

3.2.1 What to Hash

Because a good parallel NIDS should attempt to preserve connections and should direct all the packets in a single connection to a single sensor for analysis, good hashing for the purpose of parallel network intrusion detection should attempt to group packets from the same connection together.

There are many parts of the packet headers that do not vary during a connection. By using this invariant header information as the only input to a hash function, one can guarantee that all of the packets in a connection will hash to the same value. A convenient method of defining an arbitrary number of connection groups, then, is to use a hash function.

At the IP layer, flows are not well defined. However, the IP addresses of the two network hosts involved in a flow is an invariant part of the tuple of information defining a flow for any protocol that uses IP packets. The source and destination addresses occupy bits 96 through 159 of the IP packet header. Unfortunately, the IP address pair space is highly associative and clusters strongly, as demonstrated in Figure 3.2. Figure 3.2(a) is a graph of the frequency of IP addresses in the source address field, sorted by address. Figure 3.2(b) is a similar graph, but of the destination IP addresses. Both figures are from the fast trace described in Section 2.5.1.

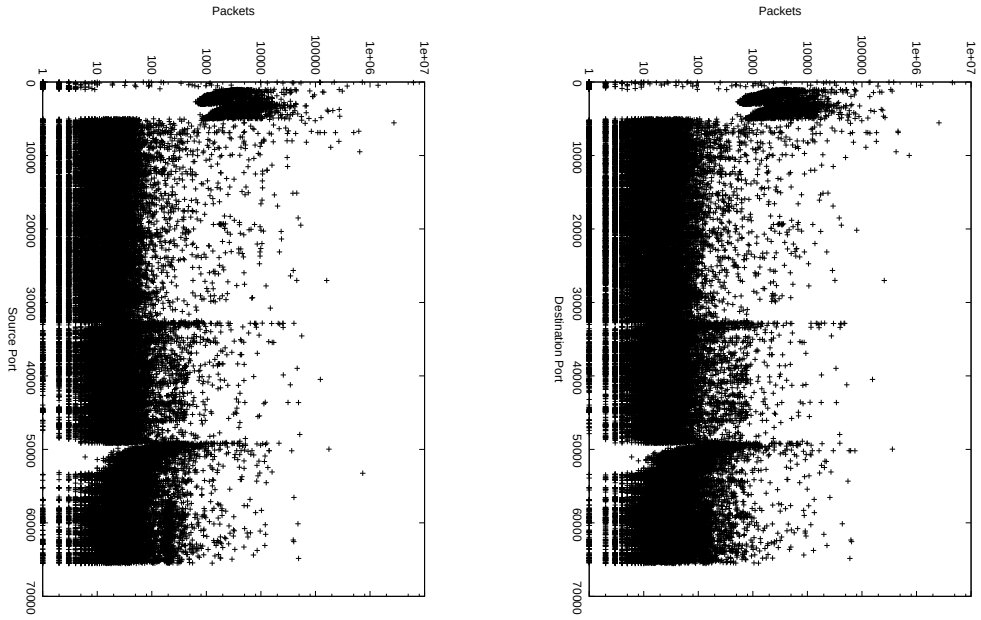


(a) Source Address (b) Destination Address

Figure 3.2. IP Address Frequency Distributions

To help offset the natural clustering of IP address tuples, more data from the packet header can and should be used, which requires further protocol-specific information. Most traffic on the Internet is either TCP/IP or UDP/IP traffic, so they make good choices of protocols to add to the hash. If new protocols were to be developed or popularized, this choice would need to be revisited. The port numbers of the two network hosts involved in the flow are an invariant part of the tuple of information defining a flow for the TCP and UDP protocols. These port numbers, in both TCP and UDP, occupy the first 32 bits of the TCP or UDP header. These bits are convenient, as they are

in the same place in both TCP/IP and UDP/IP packets, with the exception of packets including optional IP extension fields which can be accounted for easily. The distribution of the port number values observed in the fast trace is more evenly distributed than IP address values, as illustrated in Figure 3.3. Both graphs are from the same trace as the graphs in Figure 3.2.



(a) Source (b) Destination

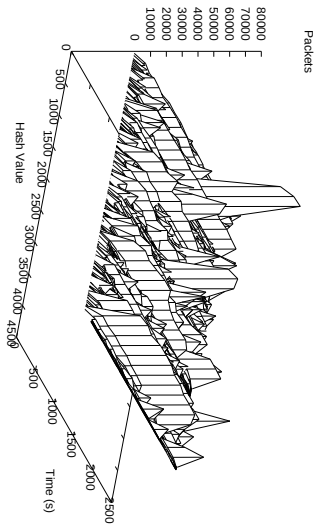
Figure 3.3. TCP/UDP Port Frequency Distributions

Not all traffic is TCP or UDP, or even IP, in nature. Knowledge of additional protocols can be added to the hash function as necessary, though there is a

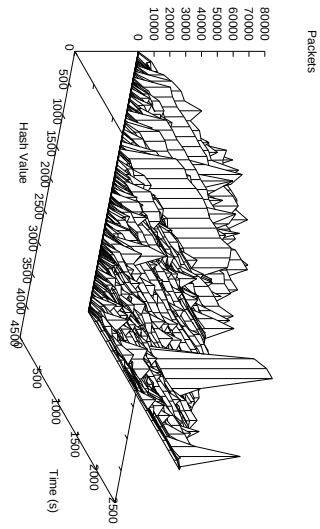
limit to how much additional protocol knowledge can be added to the hash function without losing speed. Additional bit ranges that could currently be added to the hash functions to gather still more entropy without needing to understand more protocols, are the Ethernet frame type (the 21st and 22nd bytes of the 22 byte Ethernet header) and the packet type (ICMP or IGMP, for example). However, the usefulness of additional data — particularly data that only applies to rarer packet types — is debatable. Most non-TCP/UDP protocols, like ARP, do not organize into connections and do not apply to all networks. How much additional complexity is acceptable depends on the speed of the implementation and the diversity of the traffic.

3.2.2 How to Hash

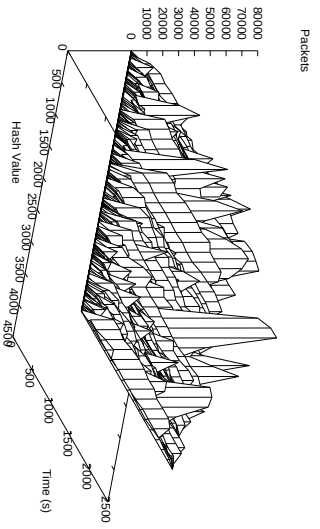
The combination of the IP source and destination addresses and the TCP/UDP source and destination port numbers provides enough entropy for several different and serviceable hashes, as demonstrated by the value distributions in Figure 3.4. Normally, the hashes are computed in parallel. Because at least one of the hash values is needed very early in the routing mechanism, there is little time for computation that requires multiple cycles. However, not all hashes are needed immediately when routing a packet. When a packet requires additional levels of hashing to determine its destination, discovering that additional levels of hashing are needed takes time. This time can be used to provide complex hash functions additional cycles to compute values if necessary.



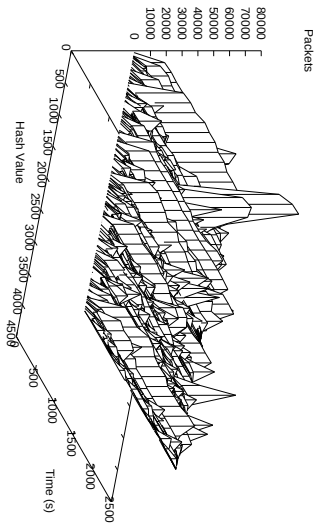
(a) Hash 1



(b) Hash 2



(c) Hash 3



(d) Hash 4

Figure 3.4. Value Distributions for Several Hash Functions

3.2.3 Bucket Power

Because network traffic density is most likely unevenly spread across the theoretical tuple-space of IP addresses and TCP/UDP port numbers, packets will be unevenly spread across the set of possible hash values. One benefit of using a lookup table to implement the mapping of hash values to sensor addresses rather than a set of ranges or a modulus operation, is that the lookup table can be modified in order to change the balance of packet load per sensor generated by the mapping. If there are more hash values than there are sensors, it is possible to shift small amounts of load from sensor to sensor to adapt to changes in network load. Larger ratios of hash values to sensors allow such adaptation to be very tailored and fine-grained. However, as the ratio of buckets to sensors increases, each hash value represents a smaller section of the tuple-space. When each hash bucket represents a small section of tuple-space, the effectiveness of a single change to the lookup table becomes correspondingly low.

The system can be tailored to a specific ratio, making a lookup table of a specific size and a specific number of sensors a requirement. Such tailoring eliminates the system's ability to scale. Adding more sensors to handle additional traffic will throw off the ratio and make the system less able to adapt to traffic load variations. In order to maintain an approximate level of effectiveness for any change to the lookup table, it is desirable to maintain an approximate ratio of lookup table entries per sensor by changing the number of lookup table entries in accordance with the number of sensors.

The prototype SPANIDS platform has a maximum routing table array size of 4,096 entries. This is adjusted down to 512 entries for less than 9 sensors,

1,024 entries for less than 17 sensors, 2,048 entries for less than 33 sensors, and 4,096 for more than 32 sensors — maintaining between 64 and 124 entries per sensor in most cases.

There has been some research recently into tailored hash functions that for a given population of input data are guaranteed to produce uniformly distributed output. While this may be possible for any given trace file or set of trace files, the nature of network traffic is highly dynamic and is therefore resistant to such static analysis. Additionally, when the NIDS is attacked, any hash function it uses can be manipulated. Only a dynamically re-balancing hash function would be able to handle the behavior of network traffic. Such a function, however, would not only be difficult to design but would also be computationally expensive and impractical for line-speed traffic routing.

3.2.4 Hashing Effectiveness

In the prototype SPANIDS, because the available memory in the FPGA is large enough to contain 4,096 routing buckets, only 12 bits are required to reference them. Thus each hash function used in the system generates a 12-bit hash value from 96 bits of input — 32 bits of IP source address, 32 bits of IP destination address, 16 bits of TCP/UDP source port, and 16 bits of TCP/UDP destination port. The 12-bit value is then scaled as necessary depending on the number of sensors.

The first two hashes used in SPANIDS are extremely simple XOR functions that combine sets of every twelfth bit of the input, starting from different bits. The first function's bit sets start at bits 0 and 1, and the second function's start at bits 2 and 4 — the selection of 2 and 4 are important, as they prevent the

second hash function from being a shifted version of the first hash function. Thus, the first of the twelve bits of output of the first hash function is the XOR of bits 0, 1, 12, 13, 24, 25, and so forth from the 96 input bits. The second of the twelve bits of output of the second hash function is the XOR of bits 1, 2, 13, 14, 25, 26, and so on from the 96 input bits. This produces a value distribution from the fast trace represented in Figure 3.4(a). The first bit of the twelve bits of output of the second hash function is an XOR of bits 2, 4, 14, 16, 26, 28 and so on from the 96 bits of input. The second bit of the twelve bits of output of the second hash function is an XOR of bits 3, 5, 15, 17, 27, 29 and so on from the 96 bits of input. This produces a value distribution from the fast trace represented in Figure 3.4(b).

The second two hashes used in SPANIDS use slightly more complex logic. Instead of taking a straightforward XOR of the 96 bits of input data, certain transformations are used on the input data first. In the third hash, for example, the 16-bit TCP/UDP port numbers are added together before being combined with the unaltered IP addresses using an XOR, similar to the first two hash functions. The value distribution for this hash on the packets in the fast trace is illustrated in Figure 3.4(c). The fourth hash function, rather than adding the TCP/UDP port numbers, adds the two 32-bit IP addresses together and then combines the sum with the unaltered TCP/UDP port numbers using an XOR similar to the first two hash functions. The value distribution for this hash on the packets in the fast trace is illustrated in Figure 3.4(d).

The hash functions discussed here were developed with the goals of being simple and easy to compute in the prototype SPANIDS system in addition to relatively evenly distributing traffic and having different collisions. Other

hash functions can be used in their place, provided they meet the same requirements. The distributions illustrated in Figure 3.4 are specific to the fast trace.

3.3 Detecting Overload

Unlike load balancers in other environments such as web servers, distributed systems, or clusters, a NIDS load balancer is not concerned with achieving the best possible distribution of work across all nodes. Since the NIDS is not in the active communication path, improving its throughput beyond the offered network load does not result in performance improvements. It is sufficient to ensure that no sensor's load exceeds its capacity.

Hashing by itself, however, is insufficient. The best hashing spreads the logical space of the network — all possible tuples of the data hashed — smoothly across a smaller set of sensors. Even a perfect hash cannot take into account the volume of traffic that may be funneled to a single hash value or group of hash values. Thus, while a good hashing algorithm evenly distributes the theoretical set of packets, it does not necessarily evenly distribute the set of packets received. Unless the incoming traffic is evenly spread over the logical space of the network, a hash function alone cannot guarantee balanced load. Something more is necessary.

3.3.1 Is There A Problem?

There are several methods available to the load balancer for determining if there is a problem with the current distribution of network traffic. If all sensors are uniform in capacity and if all packets impose identical loads on the

sensors they are sent to, it is possible to simply count how many packets or how many bytes are sent to each sensor. Then when one sensor begins to receive many more packets than the other sensors, it can reasonably be presumed that the sensor in question is being overloaded. The assumptions required to use a packet or byte counting technique, however, cannot be made.

The first assumption may not be valid. While in some circumstances it may be possible and even convenient to ensure that all sensors are identical, in many circumstances it is certainly not. Any pure load-balancer-based estimation system — whether it uses packet counts, byte counts, or some other estimation within the load balancer — must presume that all of the sensors are equally capable, which may not be the case. Each network packet imposes a certain load on the sensor it is sent to. This cost not only depends on the fixed interrupt and system call overhead, but also on the size of the packet and the actual payload. This makes it difficult for the load balancer to accurately determine sensor load based solely on the number of packets or bytes forwarded. If a sensor is being inundated by large packets, a pure packet count may be insufficient to accurately gauge the load placed on that sensor. On the other hand, while this is not modeled in this simulation, packets may contain many partial-pattern matches and thus take more time to precisely analyze, making a pure byte count insufficient as well.

The solution to these challenges is to get the sensors more actively involved in routing decisions. A simple method is to require sensors to inform the load balancer when they are receiving too many packets and may overload soon. Thus, the load balancer does not have to guess whether or not a given sensor is near overload, but instead has a much more concrete idea of exactly

how close to dropping packets each sensor is. The key to making this simple method successful is to be able to detect a trend of packets that will lead to dropped packets with sufficient advance notice that the sensor has time to notify the load balancer and the load balancer has time to compensate.

3.3.2 Locate the Problem

Once a sensor has reported imminent overload by generating a feedback packet and sending it to the load balancer, the load balancer must do something about it. A critical design decision for the load balancer is the heuristic used to determine which hash buckets are responsible for the problem that caused a sensor to generate a feedback packet. There are several viable methods.

3.3.2.1 Random

First among the methods evaluated was to simply choose random hash bucket values. This method is entirely unintuitive, but provides a good baseline for comparison to other methods. Random approaches are easy to implement and make the system's behavior difficult for attackers to predict.

3.3.2.2 Packet Counts

The second method is based on the assumption that of the buckets belonging to the sensor that sent the feedback, those that have routed the most packets are responsible for the overload. As discussed previously, counting packets is a flawed way to compare bucket "busyness" between several sensors because it makes assumptions that may not be true. Within the context

of a single sensor, however, packet counts are a simple way to determine the relative busyness of hash buckets. To implement this technique, each entry in the routing table is associated with a record of how many packets have been routed using that routing table entry. This record is reset every second. Generating a list of all hash values that belong to a given sensor every time a feedback packet is received would be very memory-intensive and may take too long. Instead, when a packet is routed, a sorted list of the most active hash values for a given sensor, or "hotlist," is updated.

3.3.2.3 Byte Counts

The third method for determining which hash values are most likely causing overload is similar to the previous method. The difference is that instead of using packet counts as the primary busyness metric, byte counts are used. The assumption of this method is that the time required to analyze each packet has more to do with overloading the sensor than the overhead of receiving each packet. The difference between this method and the previous method may only reflect the specific packet drain speed of the simulated sensors. To get a realistic picture of the difference between these two methods, more accurate drain speeds than are used in this simulation would need to be obtained.

3.3.2.4 Feedback Frequency

The fourth method for determining which hash values are most likely causing overload is based on finding hash values that have been reassigned from sensor to sensor. The idea is that if a hash bucket is frequently moved, it is

likely the cause of the problem. This method will identify such a bucket even if it has not yet received enough packets or bytes since being reset to appear very busy. The way this method works is that each entry in the routing table is associated with a record of feedback packets, much like the records in the previous two methods. When a feedback packet is received, the event is recorded in the log of all hash buckets that currently target the sensor that sent the feedback. This record is reset every second. Thus, even if a hash bucket is re-targeted to another sensor, its history of being assigned to sensors that have generated feedback packets is preserved.

3.3.2.5 Simulation Results

To compare the effectiveness of these methods, they were each simulated with both available traces using the almost entirely random default settings described in Section 2.5.3. Figure 3.5 is a graph of their respective packet-drop rates, normalized to the random mechanism and plotted on a base ten logarithmic scale. Of the 35,992,126 packets in the fast trace, using random buckets causes 284,109 packets to be dropped (0.789%), while basing the decision on feedback frequency causes 1,572,070 to be dropped (4.37%). Determining relative bucket hotness based on packets causes only 2,280 packets to be dropped (0.00485%), and using bytes as a gauge allows only 1,745 packets to be dropped (0.00633%). Results were very similar for the slow trace, where the random method had a packet loss rate of 2.04%, the feedback method had a packet loss rate of 9.51%, the packet-counting method had a loss rate of 0.0231%, and the byte-counting method had a loss rate of 0.0234%. There appears to be an extremely small difference between packet-counting and byte-

counting, however the trend is clear that using either packets or bytes is better than using either the random or feedback mechanisms. Using feedback frequency turns out to be a terrible gauge, doing much worse than a random selection. This is likely because there is insufficient feedback to make a good decision most of the time.

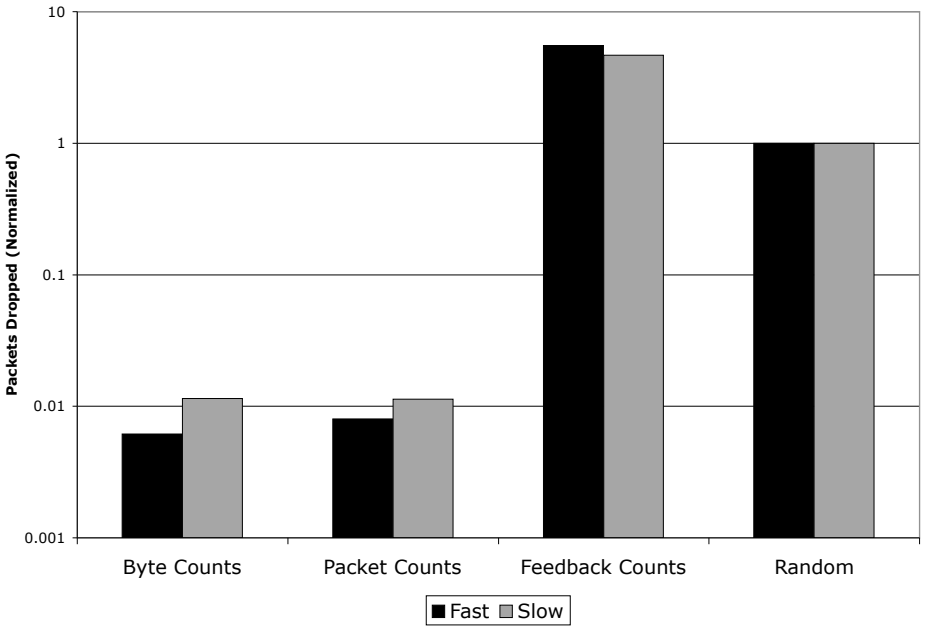


Figure 3.5. Problem Location Methods: Normalized Packet Drop Rates

3.3.3 Estimate the Size of the Problem

A second critical parameter in the load balancer design is the number of hash buckets that are affected by the load balancer's response to a feedback packet. Affecting too few buckets redirects an insufficient amount of traffic away from the overloaded sensor and thus does not reduce the sensor load sufficiently to avoid packet loss. Although the overloaded sensor can issue additional flow control messages, the sensor's packet buffer may overflow by the time the load balancer has reacted. On the other hand, moving too much traffic can overload the new sensor that receives it, and disrupts flow continuity unnecessarily. Hence, determining the correct size of the problem for remediation is important to minimize packet loss. Note that this aspect is somewhat less critical when promoting hash buckets instead of moving them, since promoting too many buckets does not negatively impact packet loss rates, but does impact flow disruption.

Figure 3.6 shows the packet loss when moving or promoting different numbers of hash buckets. These numbers are from a simulation with the default settings as described in Section 2.5.3 but with different numbers of hash buckets affected by remediation techniques. Results are normalized to the packet loss when adjusting one hash bucket at a time. A simple conclusion to draw from Figure 3.6 is that the more buckets the better: adjusting more buckets results in fewer packets lost, though there are clearly diminishing returns from additional adjustments. When hash buckets are adjusted, however, connections are broken. To get an idea of how many connections are broken when buckets are adjusted, a simple estimation technique can be used. Using the technique described in Section 2.5.4, the estimated number of broken con-

nections when affecting different numbers of hash buckets is illustrated in Figure 3.7, normalized to the number of broken connections when adjusting one hash bucket at a time.

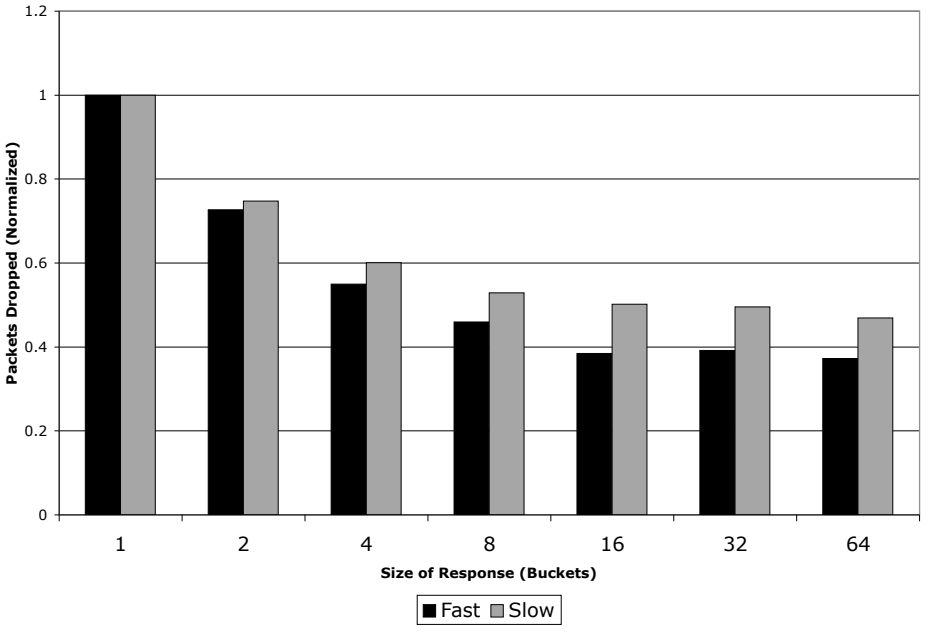


Figure 3.6. Estimated Problem Size: Normalized Packet Loss

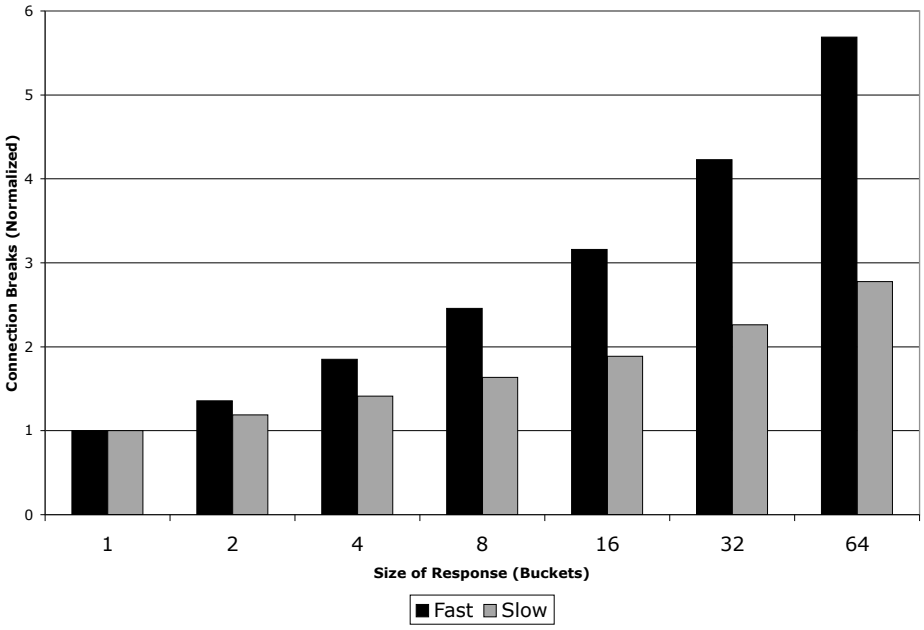


Figure 3.7. Estimated Problem Size: Normalized Connections Broken

There is obviously a tradeoff to minimizing packet loss and minimizing the number of connections broken. To explore this tradeoff, Figure 3.8 shows the weighted sum of packet loss and broken connections, normalized to the performance of a single hash bucket. In this graph, smaller numbers are better. Two overall performance measures are shown for each trace. The first metric gives packet loss a weight of 100 compared to a connection break, while the

second metric assigns a weight of 1,000 to a packet loss. The latter approach optimizes for packet loss at the expense of increased load balancer activity and more connections broken. This may for instance be desirable if the NIDS sensors employ only basic stateful analysis, and reassigning a network flow to a sensor has minimal impact on detection accuracy.

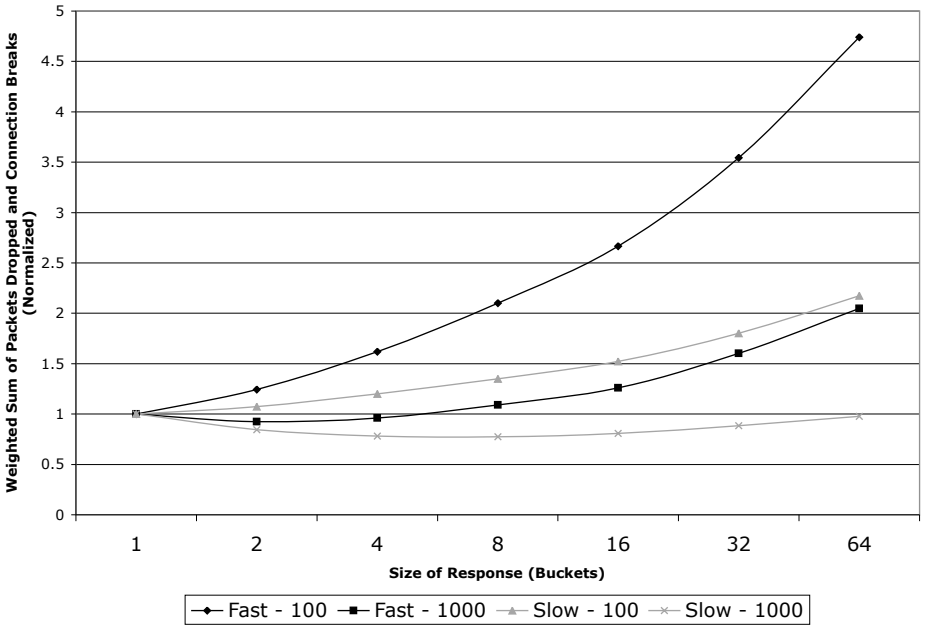


Figure 3.8. Estimated Problem Size: Performance Tradeoff

The graph shows how the tradeoff changes as the significance of packet loss changes. For a packet loss weight of 100, adjusting only one or two hash buckets provides optimal performance. On the other hand, when minimizing packet loss is considered critical, adjusting additional buckets further improves performance, particularly with the slow trace. These results suggest that the overall load balancer performance is indeed a tradeoff between packet loss and flow disruption. By adjusting the number of hash buckets that are affected when a feedback packet is received, it is possible to trade one measure for the other and thus optimize performance based on the needs of a particular installation.

3.4 Mitigating Overload

Once a problem has been reported to the load balancer and the problem-causing traffic has been discovered or decided upon, the next task for the load balancer is to address the problem. Generally, this is accomplished by directing network traffic away from the overloaded sensor and onto other less-loaded sensors.

3.4.1 Structure of Routing Table

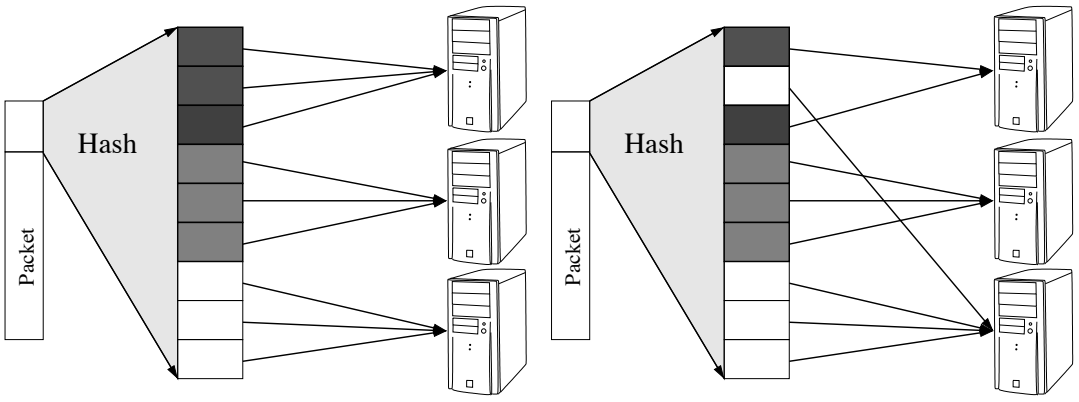
The design of the routing table must be both flexible and fast. The routing table must necessarily be altered quickly in response to feedback, and the routing algorithm for every packet must be able to operate at line speed. Therefore a hash-based lookup table is a natural design. In such a lookup table, the routing destinations for any of the hash values represented can be changed with very little interaction between the feedback-handling mecha-

nism and the routing mechanism.

3.4.1.1 What to Do With the Hash

Once an identifier for a packet group — a hash value — has been calculated for a packet, a destination sensor needs to be chosen for the packet. The hash algorithm will produce hash values within a range. In the SPANIDS prototype and the simulator, this range is between 0 and 4,096. The lookup table can be used as a one-to-one mapping function to map each hash value to a sensor.

The most simple method for associating hash values with sensor addresses is a lookup table like the one illustrated in Figure 3.9(a). To route a packet, the hash of the packet, h, must be computed based on the desired parts of the packet. The h'th entry in the lookup table will contain the address of the sensor the packet should be sent to. The entry in the array may contain more information than simply a sensor identifier, of course.



(a) Initial State (b) "Moving" a Hash Bucket

Figure 3.9. Simple Lookup Table

3.4.1.2 Splitting Hot Spots

Small amounts of traffic can be redirected away from overloaded sensors and toward another less loaded sensors by altering the lookup table. Some of the hash values that are associated with the overloaded sensor can be associated with less loaded sensors instead. This technique, illustrated in Figure 3.9(b), is referred to in this thesis as "moving" the hash buckets. This simple technique has the possible drawback of route "thrashing," where more traffic than a single sensor can handle is associated with a single hash value. Whatever sensor that hash value is assigned to will quickly become overloaded and will send feedback packets to the load balancer. These feedback packets will cause the high-traffic hash value to be reassigned again, and the cycle will repeat. This problem is a "narrow hot spot."

3.4.1.3 Narrow Hot Spots

Narrow hot spots may be detected once they have formed in any number of ways — the options are identical to the techniques outlined in Section 3.3.2. They may also be avoided entirely by preemptively addressing potential hot spots. Once a narrow hot spot or a potential narrow hot spot has been identified, the next question to answer is what to do about it. There are two methods considered by the SPANIDS load balancer. Namely, using a different hash function to split apart hash collisions, and using packet-level round-robin distribution.

3.4.1.4 Hierarchical Hashes

One way to redistribute the traffic that is mapped to a certain hash value is to hash the traffic again with a different hash algorithm in hopes that the problem is an artifact of hash collisions. For speed purposes, this likely means that multiple hash values should be computed at the same time, even if they will not all be used. The fallback hash values should be from hash functions designed not to have the same key-value collisions in order to avoid re-creating the narrow hot spot. The number of hash algorithms used is implementation dependent, however it should be noted that additional hash algorithms add complexity and time to the processing of every packet.

Several different routing tables can be used, one per hash function. Then when a narrow hot spot or potential narrow hot spot has been identified, the corresponding hash bucket can be marked so that the destination is not a sensor, but rather a different routing table. Therefore, overload avoidance mechanisms should consider the hash buckets from all of the different rout-

ing tables. A logical layout of an example two-layer scheme is presented in Figure 3.10. This can easily be expanded to more than two levels. As discussed in Section 2.2.2.4, packet-level round-robin distribution must be the last resort when flow-based distribution fails to sufficiently separate the high-bandwidth traffic that is overloading sensors.

Redirecting traffic may, however, cause subsequent parts of the flows in the affected traffic to be sent to different sensors than before the redirection. This suggests that when considering how frequently and how preemptively to use traffic redirection to correct load imbalances, there is a tradeoff between how many packets are not lost that would be lost without such adaptation and how many flows the adaptive behavior breaks across several sensors.

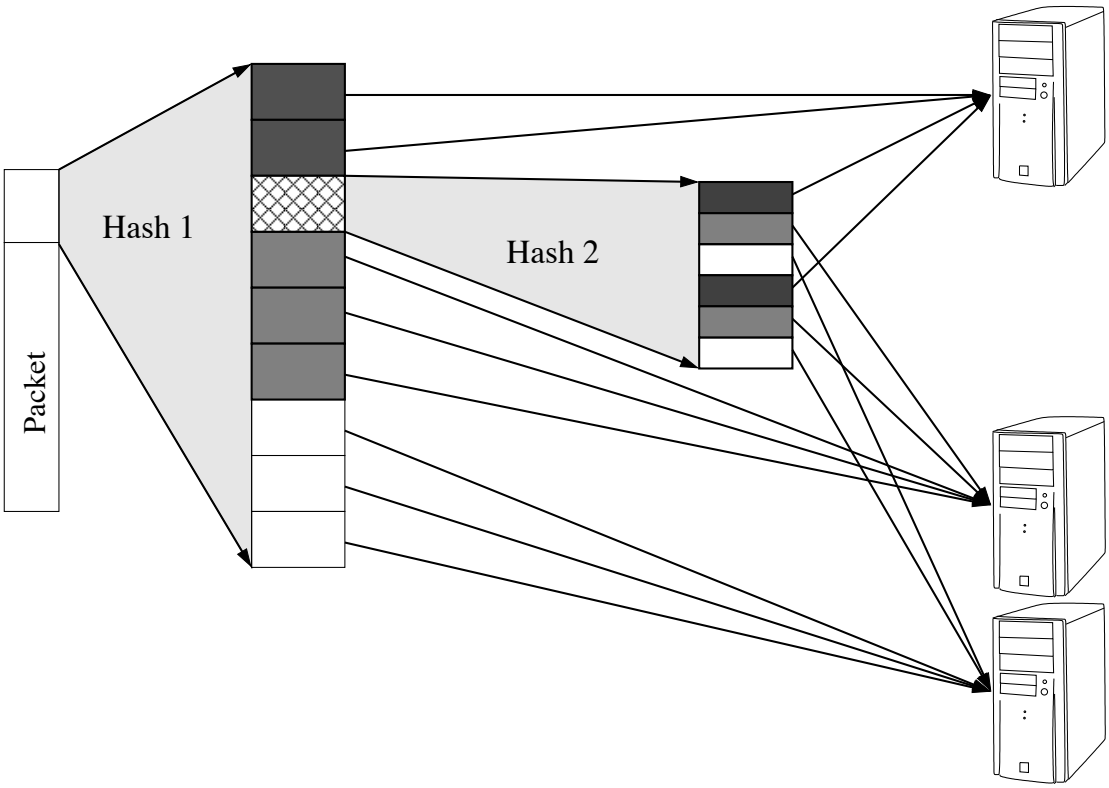


Figure 3.10. Two-Layer Hash Map Example

3.4.2 Move or Promote

The next design decision to be considered for the scalable load balancer is the heuristic used to identify narrow hot spots and thereby determine whether problem-causing hash buckets should be moved to another sensor or promoted to the next level of hashing. Neither scheme has a clear advantage over the other in terms of minimizing packet loss. However, promotion is restricted by the number of hash levels implemented in the load balancer whereas there

is no limitation on how many buckets are moved, or how often. There are several techniques that can be used to make this decision.

Static "Always Move" and "Always Promote" schemes avoid the decision of whether to move or promote hash buckets and simply apply only one of the two flow control response mechanisms.

3.4.2.1 Random

A random decision is nearly as simple as the Always Move and Always Promote heuristics. This method simply probabilistically chooses to either promote or move the buckets with a 50 percent likelihood for either alternative

3.4.2.2 Buffer Fullness

This method bases the decision whether to move or promote on the known buffer fullness of the sensor the feedback originated from. When a sensor generates a feedback packet, it can include in the body of that packet a number representing how full the sensor's packet buffer is as a percentage. In these simulations, sensors generate feedback packets when they receive a new packet and their packet buffer is more than 30 percent full. The load balancer can use a slightly higher buffer use percentage as an arbitrary line to determine whether to move or promote the problem-causing hash buckets. For the purposes of this simulation, if the buffer is more than 32.5 percent full, the load balancer decides to promote the hash buckets; otherwise the hash buckets are moved. 32.5 percent is an arbitrary line that seemed to be successful in simulation, though any cutoff can be used.

The inverse of this logic is also simulated — promoting buckets when the

sensor's buffer is less than 32.5 percent full and moving the buckets otherwise — in order to verify that the logic is correct.

3.4.2.3 Feedback Frequency and Weighted Feedback Frequency

Another useful technique for determining whether to move or promote hash buckets is to base the decision on the frequency of feedback packets. Like the method discussed for locating problem hash buckets, every hash bucket is associated with its own feedback record. Each time a feedback is received it is recorded in every bucket targeting the sensor that generated the feedback. The idea behind this technique is that while feedback frequency has been demonstrated to be a poor method of locating frequently used buckets in general, it may be a good way to identify extremely frequently used buckets that have been moved many times. The end result of this technique is that the majority of hash buckets would be moved to other sensors while particular buckets that have a record of being associated with feedback-generating sensors are promoted.

The record of feedback packets is reset every second. It may, however, be desirable to maintain some impact from history. If a decision must be made immediately after the records have been reset, all buckets appear equal: they have received no feedback packets. The feedback record can be weighted and history can be maintained by dividing the feedback count in half every second rather than setting it to zero.

3.4.2.4 Intensity and Weighted Intensity

An intuitive mechanism for deciding whether to move or promote a problem-causing hash bucket is to compare the rate of traffic assigned to the problem-causing buckets with the average rate of traffic of all buckets currently associated with the overloaded sensor. Buckets that exceed a specific relative threshold are subject to promotion instead of reassignment.

Figure 3.11 demonstrates this heuristic approach. Both graphs show an example sorted histogram of bucket load for a given sensor. The dashed line corresponds to the average load. The histogram in subfigure *a* shows a histogram with only a small variation in bucket loads. In this case it is likely sufficient to move some of the hash buckets to another sensor to address this sensor's overloading problem. The histogram in subfigure *b* exhibits a larger variation, suggesting that the network traffic associated with those few intense buckets may be primarily responsible for overloading that sensor. If that is the case, these buckets would likely overload any other sensor as well. In this scenario, the load balancer promotes the top hash buckets to the next level of hashing in an attempt to split the associated network traffic apart.

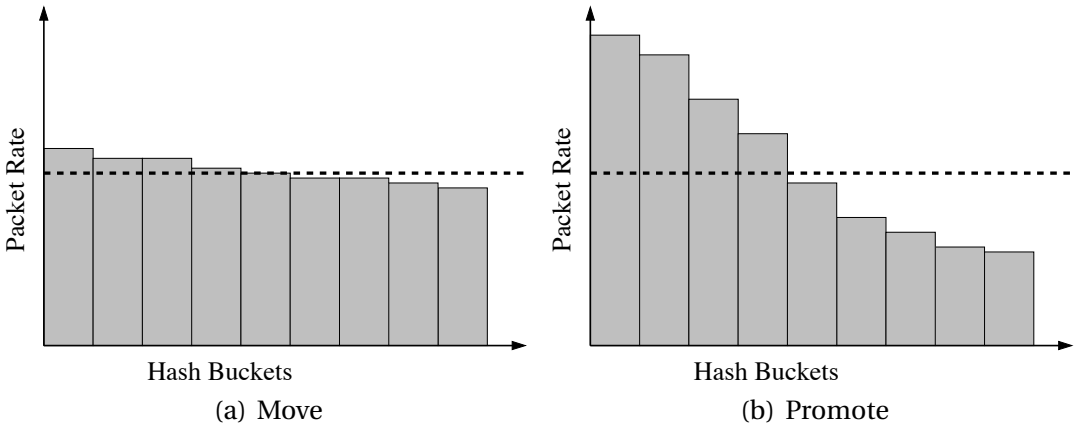


Figure 3.11. Using Intensity to Choose between Moving and Promotion

Note that this heuristic only approximates the ideal decision. First, the load balancer estimates the packet load associated with each hash bucket based on packet rates. Second, the decision whether to move or promote does not consider the shape of the histogram, but only compares the highest intensity buckets with the average load. On the other hand, this approach does not require a costly analysis of all hash buckets like a derivative of a sorted histogram would, nor does it rely on detailed and possibly outdated feedback information from the sensors. Finally, this heuristic requires little storage on the load balancer as only a small number of high-intensity hash buckets need to be tracked for each sensor.

3.4.2.5 Simulation Results

The heuristics discussed above were simulated with the two traces and the simulator's default settings. Figure 3.12 summarizes the load balancer's performance in terms of packet loss for both traces using these heuristics on a base ten logarithmic scale. The packet loss is normalized to the random method to enable a comparison between the two traces. Interestingly, the random method hits a median directly between the methods that drop large numbers of packets and the methods that drop very few packets. Also of interest, the Inverse Buffer Fullness technique outperforms the Buffer Fullness technique by a large margin, possibly indicating that the information the techniques use is frequently out of date. The lowest drop rates in both traces come from the Always Promote, Inverse Buffer Fullness, Average Intensity, and Weighted Average Intensity heuristics.

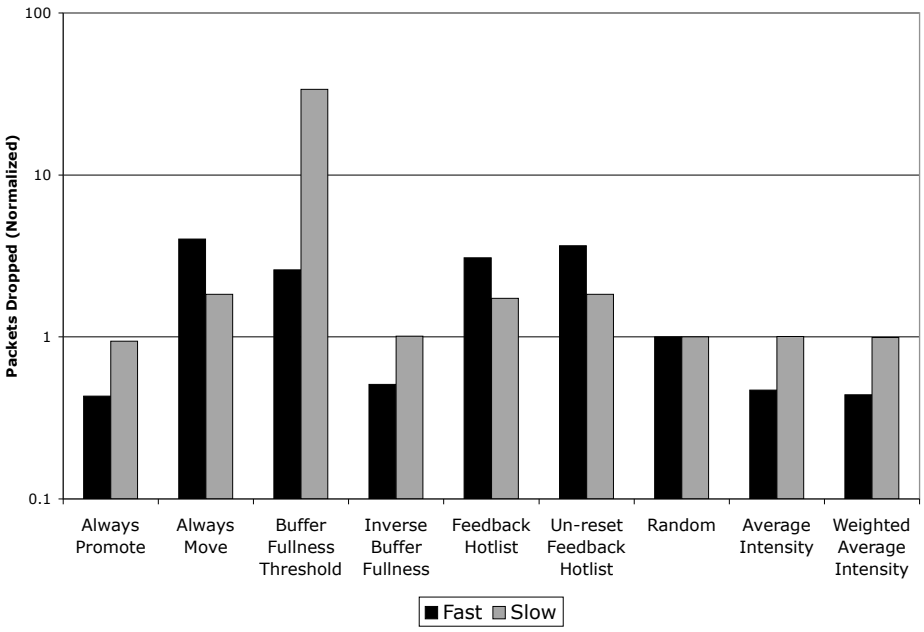


Figure 3.12. Move or Promote: Normalized Packet Loss

It is interesting, then, to compare the packet loss rates to the negative side-effects of packet loss avoidance represented in Figure 3.13. While the Always Promote, Inverse Buffer Fullness, Average Intensity, and Weighted Average Intensity methods have low drop rates, their broken connection counts are relatively high. Again, the random technique sits between the extremes. The graphs together clearly indicate that to avoid packet loss, connections must

be broken. However, to choose a good technique requires deeper analysis.

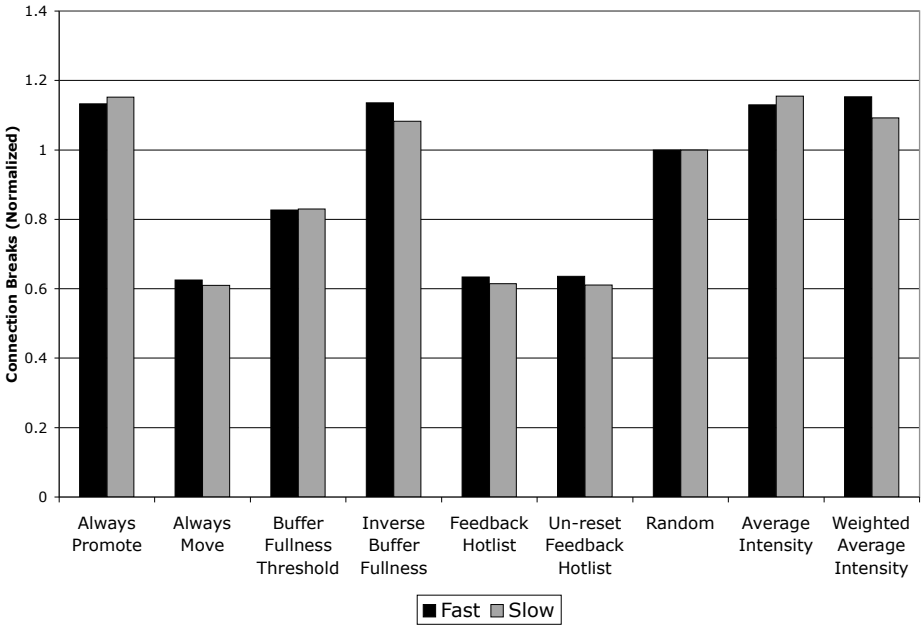


Figure 3.13. Move or Promote: Normalized Connections Broken

To explore the tradeoff between packet loss and broken connections, Figure 3.14 shows the weighted sum of packet loss and broken connections, normalized to the random method and plotted on a base ten logarithmic scale. This is similar to the previous tradeoff graph. Smaller bars are better. Inter-

estingly, when packet loss is weighted heavily the random heuristic shows the best tradeoff for the slow trace and a relatively good tradeoff for the fast trace.

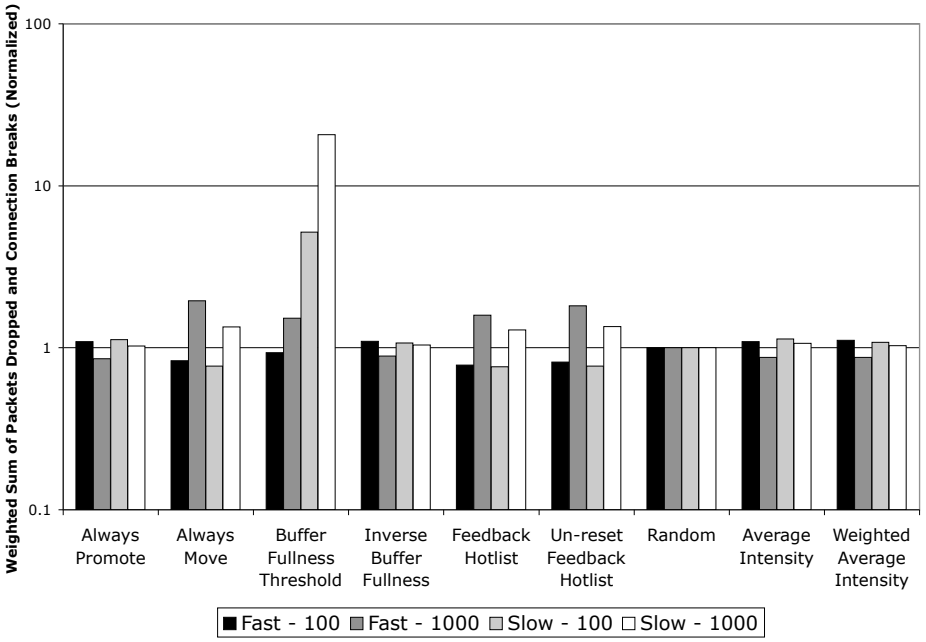


Figure 3.14. Move or Promote: Performance Tradeoff

Another metric may be useful for comparing dissimilar techniques. The success of a technique can be viewed as the number of packets that were not dropped that otherwise would have been. Specifically, the number of pack-

ets dropped if no adjustments are made to the original hash-based distribution can be used as a baseline and any fewer packets dropped when using a technique can be considered the number of packets saved. The drawbacks of a technique can be viewed as the number of connections that were broken. Thus, the number of packets not dropped per connection broken provides a measure of the effectiveness of the technique. The effectiveness of these heuristics is illustrated in Figure 3.15. Larger bars are better.

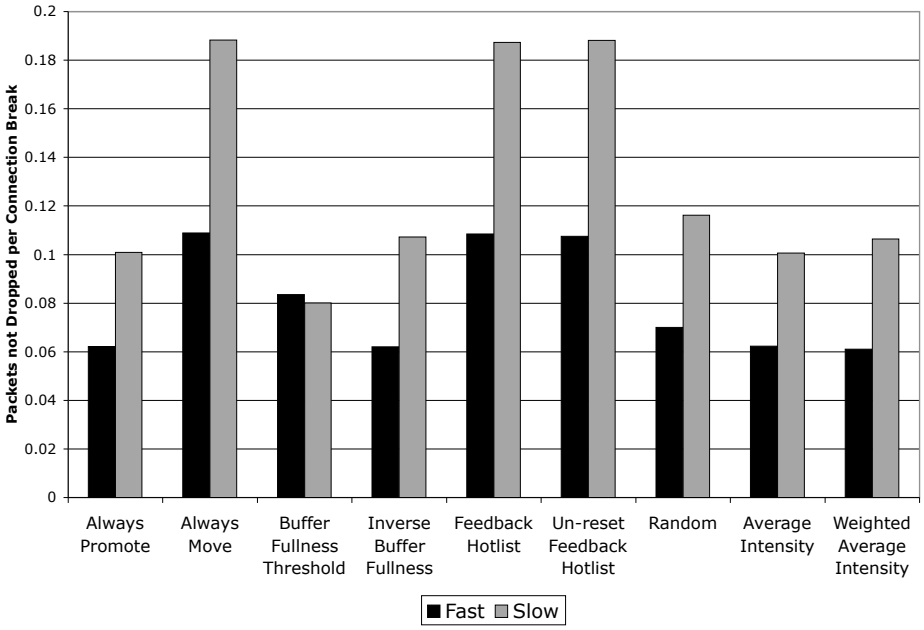


Figure 3.15. Move or Promote: Effectiveness

What is interesting about this graph is that despite the relatively high numbers of packets lost, the benefit of a very low number of connections broken outweighs the packets lost for the Always Move and feedback-based heuristics. Due to the very low number of connections broken, the feedback-based heuristics seem to have the best overall tradeoff. Even more interesting is that the feedback-based and Always Move heuristics are much more effective with the slow trace than the fast trace. This may be a result of less diverse traffic or longer connections in the slow trace, which would make the history of feedback packets more useful. In a faster trace, connections and load shift more quickly and the small delay between when the sensor generates the feedback packet and when the router receives it may make the information too old and thus less useful.

3.4.3 Where to Move

Once the decision has been made to move one or several hash buckets from one sensor to another, the problem becomes which sensor to move them to. Ideally, they should be moved to the least-busy sensor or sensors, spread according to how much free capacity those sensors have. However, the load balancer cannot know with certainty how much spare capacity a sensor has. To estimate the available capacity, the methods discussed previously for gauging sensor load may be employed. Rather than attempting to discover the most heavily loaded sensors or hash buckets, now the goal is to find the least loaded.

3.4.3.1 Random

As in previous cases, a probabilistic heuristic can be an effective tool. In other words: select a sensor at random. This technique is highly unintelligent, in the sense that it makes no pretense at gauging sensor load. It is very possible that the traffic may be re-assigned to another sensor that is operating at or near its capacity and cannot accept additional load. However, a random reassignment is far more difficult for even a careful attacker to manipulate.

3.4.3.2 Packet Counts

Just as packet counting can be used as a load determination for deciding when sensors are becoming overloaded and need load moved away, it can also be used to estimate which sensor is the least busy and could accept more load. This method has the same logical problems in this case as in previously discussed circumstances. The load balancer must presume that the sensors have identical capacity and that the number of packets is an accurate reflection of the load the packets are placing on the sensor. As before the packet records are reset every second. It is possible that a sense of the packet's historical record should be maintained, so a weighting mechanism as previously discussed is also simulated.

3.4.3.3 Feedback Frequency

Ideally, sensors can inform the load balancer of their precise capacity to accept additional traffic. To ensure such information is up to date, a two-way auction for extra traffic would need to be performed for every bucket reassignment. Such an auction would slow the load balancer's response to feedback

requests. Therefore the load balancer must base the decision on what sensor information it already has, namely the feedback that they have sent. Directing the traffic to the sensor that has sent feedback to report overload the least is an easy method for locating a sensor with available capacity based upon sensor-provided data. Counts of how many feedback packets a sensor has generated may suffer from the same problems counts of how many packets are routed to each sensor do as detailed above. Thus, a counter weighted in the same manner is also simulated.

3.4.3.4 Simulation Results

The heuristics discussed were simulated using both traces and the "average" technique for the decision to move or promote buckets. Figure 3.16 summarizes load balancer performance in terms of packet loss. Packet loss is normalized to the random method to enable a comparison between the two traces. Interestingly, random is one of the more successful methods. Using packet counts and weighted packet counts produces results similar to the random method and the two techniques are reasonably consistent between the two traces. The feedback-based methods appear comparable to the random heuristic when used for the fast trace, but have surprisingly high packet loss rates for the slow trace.

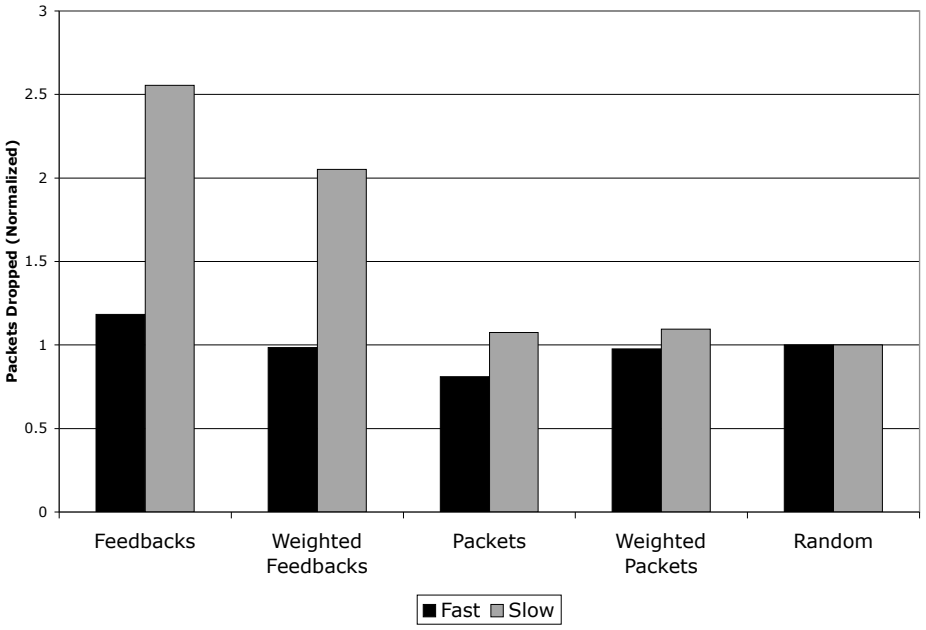


Figure 3.16. Move Target Method: Normalized Packet Loss

Looking at the methods in terms of connection breaks, illustrated in Figure 3.17, is also interesting. The feedback-based methods have both more broken connections and greater packet loss than the random heuristic. At the same time, the packet-based methods are very similar to the random method, though with slightly fewer connection breaks, and are rather consistent between the two traces.

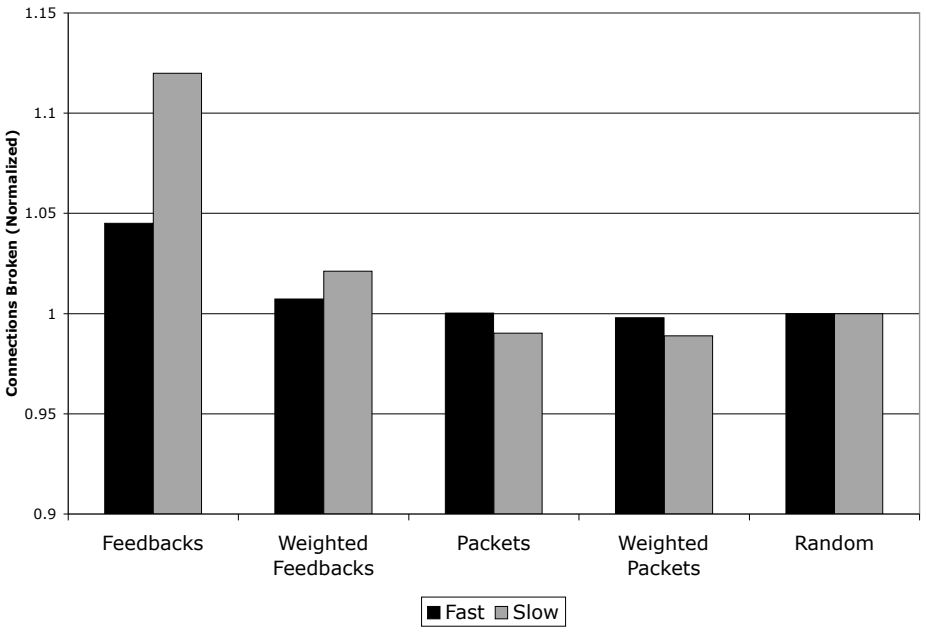


Figure 3.17. Move Target Method: Normalized Connections Broken

The effectiveness metric discussed previously — the ratio of packets saved to connections broken — is illustrated in Figure 3.18. The ratio is normalized to the random technique to allow for comparison between the traces. This graph makes it clear that random is an excellent technique, though the weighted packet heuristic has a slightly better ratio of packets saved to connections broken.

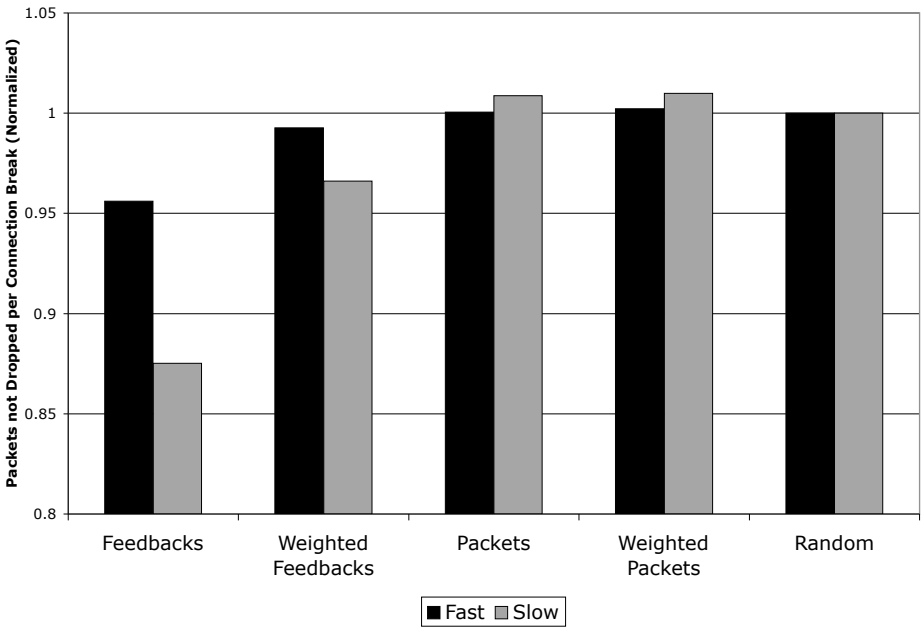


Figure 3.18. Move Target Method: Effectiveness

3.4.4 Coalescing

The final issue to consider when dealing with narrow hot spots is how to revert to the way things were originally once the hot spot has cooled down. Multiple levels of hashing can be very useful for dealing with NDoS attacks. However, unless the additional layers are temporary and are collapsed as the hot spots cool, eventually all traffic will be filtered through all hashing layers.

Thus, the traffic will be distributed solely according to the last hash layer — or ultimately, the round-robin fall back — which defeats the entire purpose of using multiple hashes.

One possible way to determine when to collapse, or "demote," the hashing layers is similar to the way the hashing layers are expanded: based on traffic. Packet counts and collective feedback counts can be useful for this purpose because they can be associated with specific hash values in specific routing tables and are not restricted to a purely per-sensor use.

A somewhat simpler method for coalescing, and the one the SPANIDS project uses, is a timeout on bucket promotion. When a hash value receives enough traffic to promote it to another level of hashing, it receives a counter value that is decremented every second. When the counter value reaches zero, the hash value's promotion is re-evaluated. If the bucket would be promoted again it receives a new timer value, otherwise it is demoted. When using such timers, the timer should be a random value in order to avoid making assumptions about the sensor and the traffic patterns. A random timer also decreases the predictability of the system.

3.5 Generating Feedback

3.5.1 Being Agnostic

The last of the four major critical areas of the design of a scalable parallel NIDS platform is located in the sensors. This is a difficult area in which to dictate design because one of the goals of SPANIDS is to be as detector agnostic as possible. That is, the technique and implementation of the traffic analysis software used on the nodes should not matter to the load balancer and

preexisting software should work without modification. On the other hand, SPANIDS is addressing a capacity problem and must rely on the cooperation of the sensors to give the load balancer some knowledge of the capacity the sensor nodes possess. To that end, a compromise can be reached. Each sensor has a responsibility to the rest of the system: to generate feedback packets when it is nearing overload. This responsibility can be handled without the knowledge of the traffic analysis software running on the node. NIDS software typically uses a RAW network socket to receive packets from the network. In the case of the prototype system, the sensor nodes are all Linux systems and the self-monitoring and participation in the SPANIDS platform is handled by a custom kernel module implementing the RAW socket interface. This kernel module monitors the RAW socket's buffer, and can generate feedback packets as it deems necessary. This process is invisible to the NIDS traffic analysis software on the Linux hosts.

3.5.2 Techniques

3.5.2.1 Fullness Threshold

There are many techniques that can be employed to decide when to send a feedback packet. The simplest method is a threshold. Every time a packet arrives, the sensor compares the cumulative size of packets in the buffer with the full size of the buffer. An arbitrary threshold of 30 percent was chosen, such that if the buffer is more than 30% full a feedback packet is sent to the load balancer to report that the sensor is becoming overloaded.

3.5.2.2 Fullness Threshold with a Rate Limit

After a sensor sends a feedback packet to the load balancer and the load balancer compensates, the sensor will continue to receive new packets to process. However, the sensor's buffer is not instantly emptied, and may still be over the threshold as these new packets arrive, and may cause additional feedback packets to be sent to the load balancer even though appropriate corrective action has already been taken by the load balancer. Thus there is a tendency in the system to overcompensate. Every time a sensor's buffer exceeds its threshold, it has a tendency to send several feedback packets to the load balancer in quick succession while the system re-stabilizes.

A simple fix to the overcompensation problem is to limit the rate feedback packets may be generated. While it is desirable to allow the sensor to continue to generate feedback packets to indicate to the load balancer that the problem has not yet been resolved, the load balancer's corrective action must be given time to take effect. Time is unfortunately in limited supply, as the sensor's packet buffer is finite. Since the feedback threshold is 30% there is some time available to see if the corrective action has had a sufficient impact, but not much.

3.5.2.3 Packet Arrival Rate Threshold

It is possible for the buffer to fill to approximately 30% (or any other arbitrary threshold) and remain there as packets are removed at approximately the same rate as new packets arrive. In that case, being above the 30 percent threshold does not necessarily indicate that the sensor is in danger of losing packets. Sending feedback to the load balancer in such a situation may dis-

rupt flows unnecessarily. This is true of any arbitrary threshold.

A more intelligent method may be to generate a feedback packet only when the rate of change in the buffer use breaches a certain threshold. This presumes that slowly increasing buffer use is unlikely to be a problem. This technique can be combined with the previous technique, allowing the packet buffer to slowly fill to as much as 60 percent full without complaint, while still generating feedback even below that 60 percent threshold if the buffer starts to fill too quickly.

3.5.2.4 Predicting the Future

Even a slowly filling buffer will eventually overflow. Additionally, allowing the buffer to get too full before generating feedback means that the sensor may not be able to provide the load balancer with sufficient notice before dropping packets. Finally, a quick jump in buffer fullness when the buffer is nearly empty may not indicate that the sensor is in danger of losing packets and generating feedback may disrupt flows unnecessarily.

A comprehensive mechanism must predict the future. Rather than rely on an arbitrary threshold, the rate the buffer is filling can be used to predict how much time the sensor has before packets are lost. If the estimated amount of time before the sensor's buffer becomes full is lower than a given threshold, a feedback packet is sent to the load balancer. This allows slowly filling buffers to fill without concern, and quickly filling buffers that are mostly empty to cause no alarm, as long as there appears to be no danger of imminent packet loss. In the simulation, predictions are made every hundredth of a second (10 milliseconds), and estimate buffer use a specific multiple of that length of

time into the future.

3.5.2.5 Simulation Results

The packet loss associated with several timeout values, rate thresholds, and predictive estimates are compared in Figure 3.19. These feedback mechanisms are compared by connections broken in Figure 3.20. Both graphs are normalized to the Always Report 30% threshold technique described above. The packet loss is plotted on a base ten logarithmic scale. It can be seen that any limitation on the speed that feedback packets are generated causes more packets to be lost than the Always Report technique. However, the Always Report technique breaks more connections than any of the techniques that limit feedback generation speed. It is interesting to note the large increase in connections broken and large decrease in packets dropped between the rate limit of one feedback packet per second and ten feedback packets per second. This may indicate that the sensors are frequently overloaded in under a tenth of a second. It is also interesting to note that the predictive feedback generator must predict a tenth of a second into the future in order to get a packet loss rate lower than the Always Report technique. However, if the predictive feedback generator predicts more than seven hundredths of a second into the future, it breaks more connections than the Always Report technique. Most likely, this is a result of incorrect predictions resulting in more feedback packets generated than necessary. It may be tempting to look favorably upon the surprisingly low level of broken connections caused by the percent change threshold techniques. Deeper analysis of the simulation results reveals that this is due to the extremely high packet losses — large numbers of connec-

tions were never received by any sensor, and thus could not be broken across several sensors.

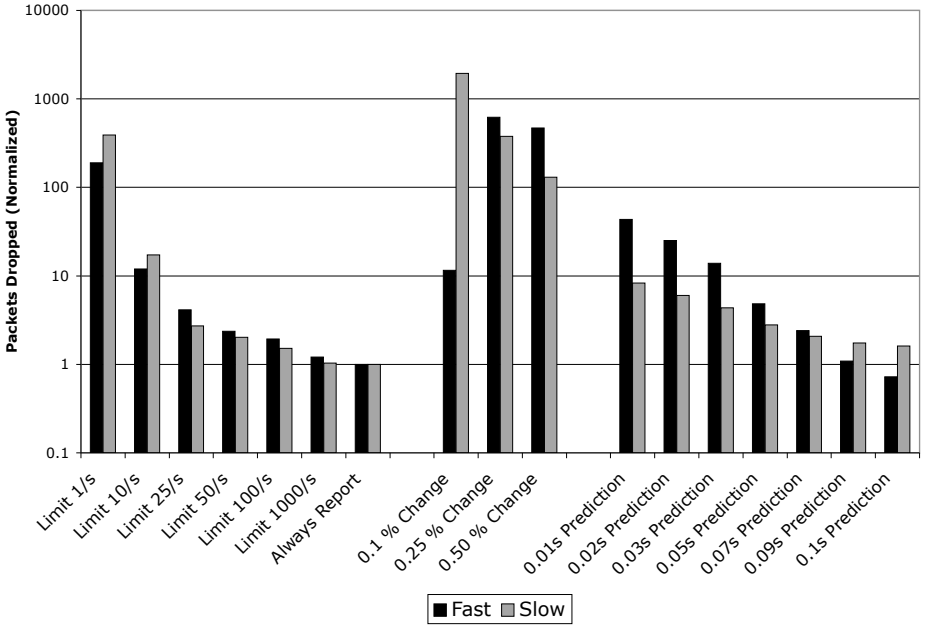


Figure 3.19. Feedback: Normalized Packet Loss

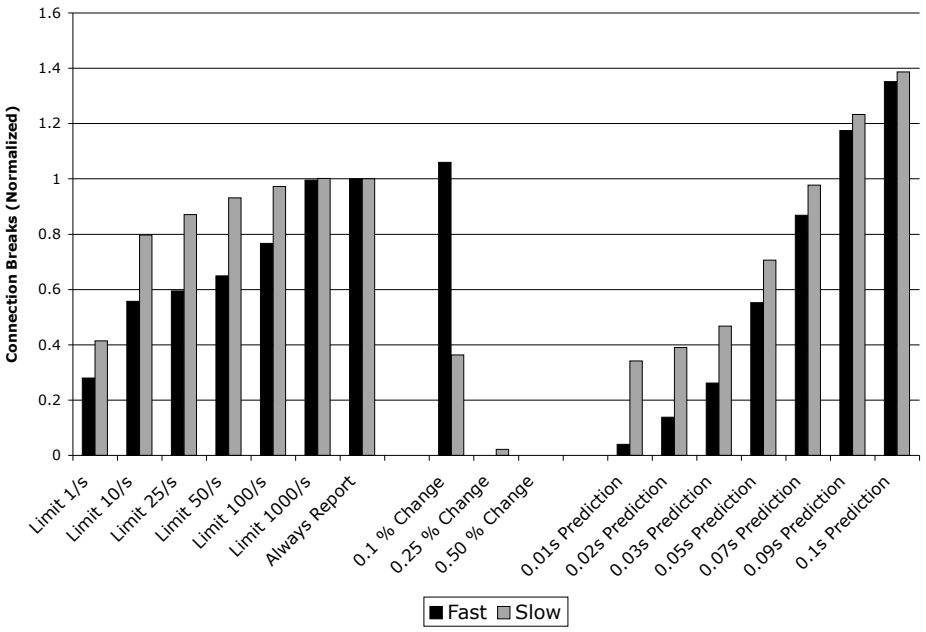


Figure 3.20. Feedback: Normalized Connections Broken

The effectiveness metric described in previous sections is useful for comparing these techniques further. The effectiveness of limiting the rate that feedback packets can be generated is presented in Figure 3.21. The effectiveness of using the current packet rate to predict the future is presented in Figure 3.22 and the same metric when generating feedback packets based on the speed the buffer fills is presented in Figure 3.23. The data point for limit-

ing feedback to one feedback packet per second when using the slow trace is missing from Figure 3.21 — the data point (-4.82) is sufficiently negative that to display it would compress the detail out of the rest of the graph.

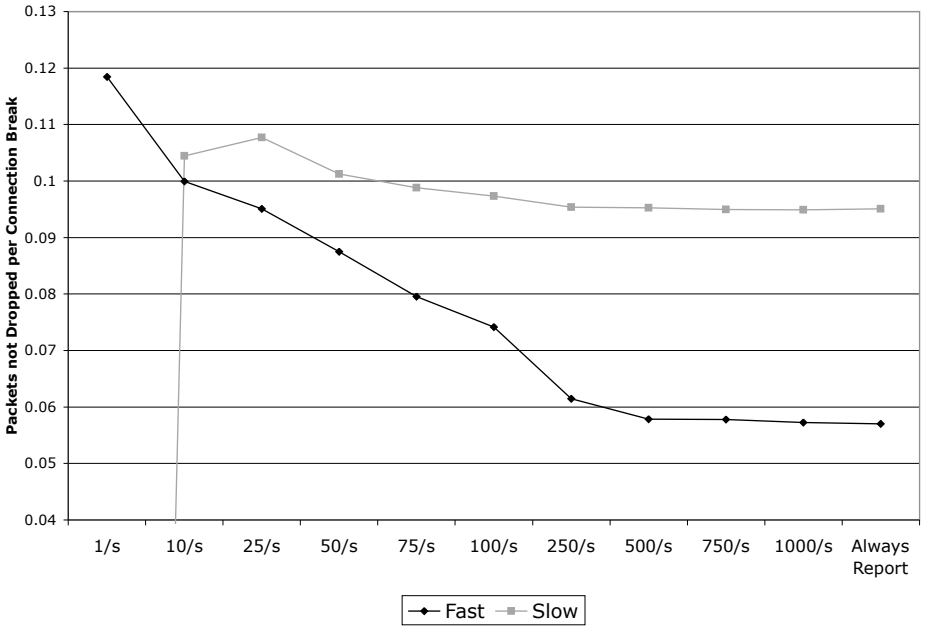


Figure 3.21. Feedback: Threshold Rate Limit: Effectiveness

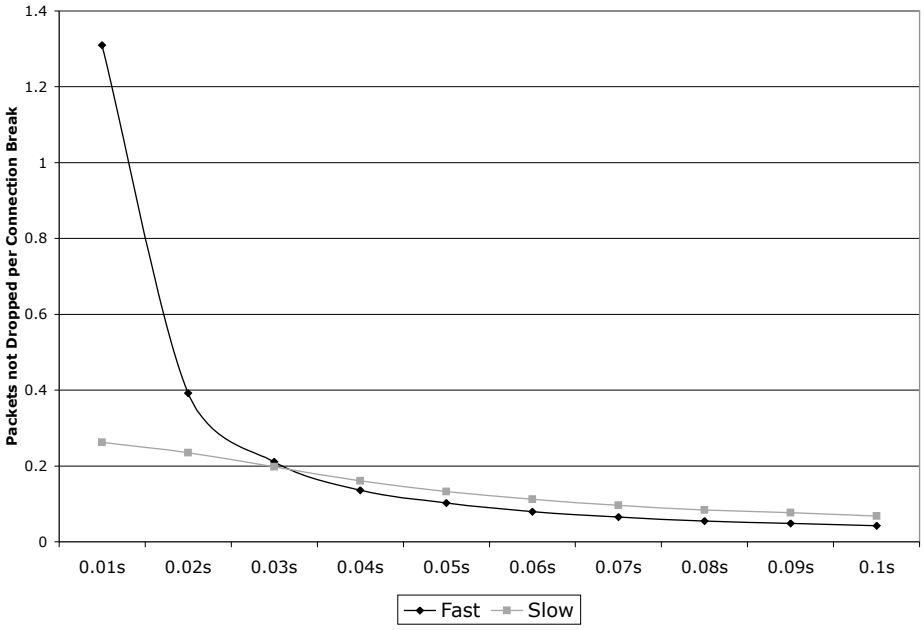


Figure 3.22. Feedback: Prediction: Effectiveness

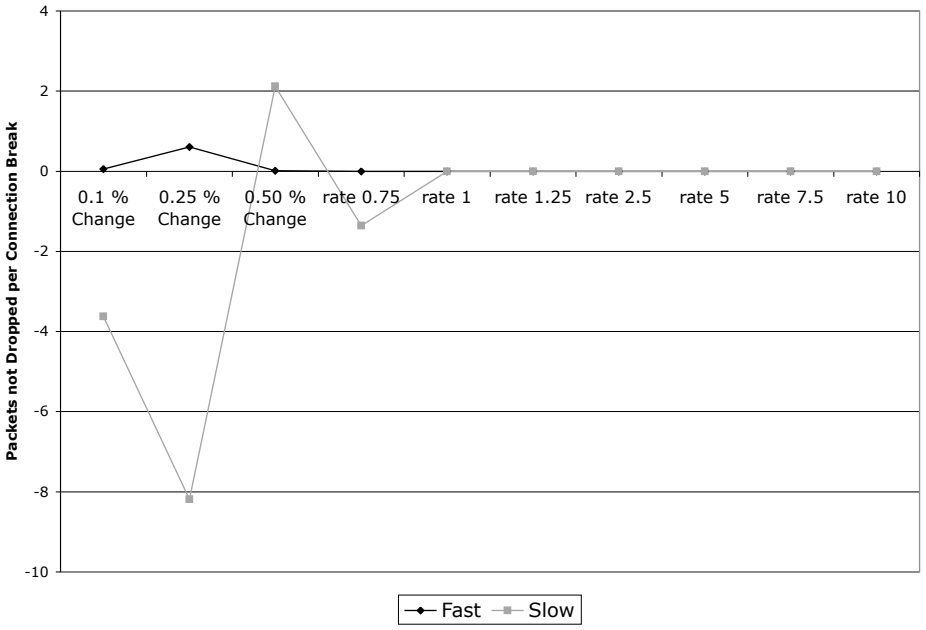


Figure 3.23. Feedback: Rate Threshold: Effectiveness

Predictions about the very near future, as displayed in Figure 3.22, are very effective at preventing unnecessary feedback packets and thus unnecessarily broken connections. It is interesting to note that the most effective rate limit for rate-limited feedback packet generation is at approximately 10 or 25 feedback packets per second for the slow trace, further suggesting that sensors are overloaded most frequently in approximately a tenth of a second. The

difference between the two traces in effectiveness when using the predictive technique is particularly striking. The difference suggests that the speed of the fast trace overwhelms sensors more quickly than the sensor can report it, making prediction and early detection more much more useful. Perhaps the most intriguing result is that in several cases, particularly the cases that use the packet buffer filling rate, the effectiveness metric is negative. Each instance of a negative effectiveness metric is a result of more packets being dropped than the baseline of no feedback packets generated at all. The irregularity of the rate threshold technique's effectiveness is also caused by high packet loss rates. These techniques demonstrate the power of misused feedback packets to direct traffic away from less loaded sensors and onto heavily loaded sensors.

3.6 Summary

The questions posed at the beginning of this chapter to outline the critical areas of design in SPANIDS can now be answered easily. The questions were: how should traffic be divided up, if there is a problem what traffic is causing the problem, how should the problem be resolved, and how does a sensor know if there is a problem?

The traffic is divided using an XOR-based hash function of the TCP/IP and UDP/IP headers that define TCP and UDP traffic flows. The hash value produced is used as the index into a lookup table that maps the hash values to sensor addresses. Several different hash functions can be used as sequential backup hashes to separate hash collisions when large traffic flows hash to the same value. The SPANIDS prototype uses four hash functions with an equal

number of associated lookup tables.

If there is a problem, the traffic that is causing the problem can be identified by using packet counts. Within the context of the set of hash buckets associated with the sensor that generated a feedback packet, packet counts easily identify frequently used hash values. There is a tradeoff between the number of hash buckets that are considered part of the problem and the amount of dropped packets and connections broken. Essentially, when more hash buckets are considered part of the problem, the response to a feedback is larger and more traffic is directed away from the overloaded sensor, reducing packet loss. However, when more hash buckets are affected there is a greater likelihood that the effort to avoid overload is too large, breaking connections unnecessarily. This tradeoff is well behaved, and can be tailored to the capabilities and requirements of the NIDS software deployed on the sensors.

Once the problem-causing hash buckets have been identified by the load balancer, they can be either assigned to a different sensor or distributed using a different hash function. Whether to change hash functions or reassign the hash buckets depends on whether the hash buckets can overload any sensor they are assigned to. The most effective method of estimating this is to rely on the bucket's history of assignment; if the hash bucket has been assigned to sensors that have generated feedback it is more likely to be the cause of the overload. However, choosing randomly is surprisingly effective as well. If a hash bucket must be reassigned, a new sensor must be chosen. Assigning the bucket to the sensor with the smallest value of a weighted packet record is the most effective solution, though a random choice is extremely effective as well.

Finally, a sensor can monitor its packet buffer to determine whether it is

becoming overloaded. Of the several techniques examined, the most effective technique is to use the current rate the sensor's packet buffer is filling to predict a tenth of a second into the future whether the sensor will be overloaded. Mispredictions can cause unnecessary connection breaks, but the large number of packets not dropped is worth the broken connections.

It is interesting to note that for most of the simulation results, the two different trace files that are used produced very similar data despite an order of magnitude difference in speed and despite covering different very different lengths of time. This general similarity is encouraging, as it indicates that the techniques employed here do not generally rely on the speed of the traffic being monitored. Where the results do differ, the cause is most likely the difference in speed. The difference is most obvious in the few techniques that rely on specific timing.

CHAPTER 4

RELATED WORK

4.1 General Load Balancing

The distribution of network traffic over clusters of nodes is commonly used in network services such as web servers [7, 10]. Such load balancers often maintain per-flow state to ensure that connections are not disrupted, or use a static hashing approach.

An example of the static hashing approach is the ONE-IP project [7], which suggests that each server in a cluster of servers providing a scalable service should take responsibility for a range of hash values from a hash based on IP source addresses. Network traffic is broadcast to all nodes in the cluster. Each node's network driver ignores traffic from IP addresses that do not hash to within that node's assigned range. This approach assumes that the requesting hosts hash relatively evenly and that such static assignment is sufficient to handle network load variations. However, the technique is only meant to provide a theoretically higher capacity, not adapt to changes in load. Similarly, the ONE-IP broadcast technique does not address problems that may result from interrupt and system-call overhead.

An example of the per-flow state approach is the TranSend system described by Fox et al [10]. In this system, each incoming connection is han-

dled by a thread in the TranSend front-end server. This server then behaves similarly to a proxy server. It chooses a back-end node to handle the request, makes the request, and returns the response to the original requester. The TranSend approach assumes that the overhead of servicing a request is far greater than the network communication overhead. This approach is acceptable in the TranSend system and similar systems since it is possible to reject new connection requests. A NIDS, on the other hand, is not part of the network conversation and has no means to throttle network traffic or reject new connection requests. Therefore a NIDS must be able to handle the maximum load that can be sent via the monitored network link. Additionally, a TranSend-like system is easily overloaded. Fox suggests running a TranSend front-end with a mere 400 threads, allowing only 400 simultaneous connections, which is several orders of magnitude too small to be resistant to a denial of service attack.

4.2 Parallel NIDS

Previous work in parallel network intrusion detection has also employed flow-based distribution strategies [9, 18]. Similar to conventional routers, these systems maintain tables of established connections to route packets to the appropriate sensor nodes.

The Kruegel approach uses either a statically assigned mechanism for determining where to send traffic, or a statically assigned mechanism for determining when to redirect flows — both of which require full connection tracking. Neither method alters its routing based on the real load of any of the sensors [18].

The TopLayer system attempts to avoid the re-assignment problem altogether [9]. The TopLayer system has a stateful flow tracking mechanism in its load balancer, and flows are assigned to the sensors in a round-robin fashion. Because it is possible for packets to be lost even when using flow-based distribution, TopLayer duplicates flows, ensuring that each flow is sent to at least two different sensors. The hope is that at least one of the sensors will successfully receive all of the packets in the flow. This approach supports flow-based intrusion analysis at the sensors, and even protects sensors from simple alert-based overloading [11] but makes the overall system vulnerable to SYN-based denial-of-service attacks. The SPANIDS architecture does not exhibit this scalability limitation. Additionally, neither the TopLayer approach, the Kruegel approach, nor other existing systems use dynamic feedback to improve system robustness and adaptability.

4.3 NIDS Technology

Network intrusion detection is an active field of research, constantly developing new approaches and techniques. The SPANIDS project leverages these improvements by using off-the-shelf sensor hardware and software. The scalable load balancing approach provides a significant capacity improvement, independent of and orthogonal to any improvements in the NIDS sensor software or hardware. While the prototype system uses Snort [13, 31], it is not the only approach that can be used with the SPANIDS platform. The SPANIDS architecture is designed to be NIDS-agnostic, capable of working well with any detection technology. Snort performs signature-based analysis of network traffic and only known malicious traffic can be detected — a popular

technique [6, 22, 26]. The alternatives include stateful flow analysis [28] and statistical anomaly detection [24, 33]. Stateful flow analysis carefully defines the state transitions and logical flow of valid network communication. Variations are considered malicious. Statistical anomaly detection [24, 33] makes the assumption that attackers behave differently from ordinary network users and therefore any statistically unusual behavior must be malicious.

The SPANIDS system assumes the classical NIDS approach of network-invisibility and noninterference. Some projects, like Hogwash [19] and Shield [34], take a more intrusive approach of blocking traffic that is recognized as malicious. Because packets that travel through SPANIDS are not altered significantly by the load balancer, it would be possible to adapt SPANIDS to work with such intrusive systems.

4.4 NIDS Performance Analysis

Network intrusion detection performance has been measured extensively, both in terms of capabilities [12, 21, 29] and capacity [20, 30, 32].

Puketza lays out three objectives for a quality IDS: broad detection range, economy in resource usage, and resilience to stress [29]. These objectives are then estimated by running sets of many different automated attack scripts against target hosts to determine how well the intrusion attempts are detected. Puketza uses network traffic as one of the stressful situations an intrusion detection system can be exposed to that decreases its effectiveness. In 1999, a large network intrusion detection test was run at Lincoln Laboratories to determine the qualitative success of the available analysis methods [12, 21]. The effort coordinated efforts between a wide variety of research groups and con-

tributed a great deal to standardizing metrics of NIDS success. The test revealed many problems with NIDS detection techniques, including difficulty with detecting new forms of attack, difficulty with detecting complex attacks, and high false positive rates.

The work presented here is motivated by the realization that increasing network speeds make intrusion detection systems vulnerable to overload scenarios that are not related to the specific analysis techniques used. Schaelicke demonstrated that commodity hardware is incapable of sufficiently handling even the top network speeds of ten years ago [32]. More critically, in Schaelicke's tests newer processors such as the Pentium IV were less capable of handling the network load than older processors like the Pentium III. This lack of improvement indicates that the inability to handle the network load is unlikely to be solved by newer and faster processor technology. Were the problem directly related to the complexity or efficiency of the packet analysis algorithms, the faster Pentium IV should have offered large improvements over the Pentium III. The SPANIDS project presented here directly addresses the capacity bottleneck, while also developing and refining methods to evaluate NIDS performance in terms of capacity. Though the nature of the quantitative (capacity) and qualitative metrics and design space appear orthogonal, improvements in capacity allow more resources to be devoted to complex algorithms that improve the qualitative success of intrusion detection systems.

CHAPTER 5

CONCLUSIONS & FUTURE WORK

5.1 Future Work

There are many avenues left to explore in the realm of NIDS load balancing. Future work includes the design and investigation of techniques to make the load balancing heuristics less deterministic and predictable, reducing the vulnerability of the NIDS platform to sophisticated evasion techniques, as well as better fallback techniques. Other avenues for future work include the development of techniques to transfer flow information maintained by individual NIDS sensors in the event that the load balancer moves a flow to another sensor. Furthermore, the correlation of observed events will become more important and parallel NIDS architectures must include light-weight communication mechanisms to facilitate the exchange of such information.

5.1.1 Randomization

5.1.1.1 The Benefits of Random

Predictability is generally undesirable in a secure system like a NIDS load balancer. If the load balancer can be affected in a predictable way, it is possible for a sophisticated attacker to carefully plan network events that will cause the load balancer to overload a sensor. Probabilistic behavior reduces the

causal link between network events and specific responses. For example, it is much harder for an attacker to cause the load balancer to focus traffic on a given sensor when the assignment of traffic to sensors is unpredictable and unrelated to any input the attacker may provide.

5.1.1.2 New Applications of Random

There are several ways that additional randomness may be introduced to the system to reduce overall predictability without loss of effectiveness. For instance, the load balancer may randomly choose one of several hash functions when promoting hash buckets to the next level. This would make it very difficult for an attacker to predict the precise result of a promotion.

Another example of an area that can be modified to be more probabilistic and less predictable is problem size estimation, described in Section 3.3.3. In that section, the effect of treating overload problems as the result of one of several preset problem sizes is detailed. It may be useful to choose a random number of hash buckets to move or promote instead of using a preset problem size. Using random problem sizes may provide a better tradeoff between dropped packets and broken connections than a static problem size assumption. Additionally, even a very knowledgeable attacker cannot know or affect how many hash buckets will be used to resolve sensor overload when the number is chosen randomly.

5.1.1.3 Alternative Probabilities

Several of the techniques detailed in Chapter 3 already use random numbers to make decisions. In each case, the probability that any one alternative

will be chosen is the same as any other alternative. The probabilities used for each decision alternative do not need to be equal. For example, when randomly deciding whether to move or promote a hash bucket it might be beneficial to choose to move rather than promote 60% of the time, or 40% of the time. Other random methods detailed in Chapter 3 may benefit from similar alteration.

5.1.2 Problem Size

One area of the design space that could be explored further is the method to estimate the size of the problem that is causing sensor overload. Currently, a static number of hash buckets are used to avert sensor overload when a feedback packet is received. It is likely useful to dynamically estimate the problem size. Such an estimate could be based on many of the same techniques that have been used to determine whether to move or promote problem-causing hash buckets. Such techniques would most likely be much more resource intensive than the current static method, but the improvement in packet loss and broken connection levels may be worth the additional resources.

5.1.3 Round Robin

The round robin packet distribution technique is the method most taken the for granted in this thesis. The round robin technique as implemented in the simulator is a rotating route path that iterates to the next sensor every time it is used to route a packet. This is a simplistic implementation of a round-robin distribution scheme. The method is used as the last resort when a flow overloads a sensor and cannot be split apart by the available hash functions.

However, this use of the round-robin technique is not fully justified. While the technique does evenly distribute packets, it does not account for sensor heterogeneity, packet heterogeneity, or known sensor load. Thus, its interaction with other techniques for load balancing is neither well-defined nor well-explored. Alternative fallback mechanisms should be developed.

5.2 Conclusions

Network intrusion detection is one of several important security measures commonly employed to secure critical data. As such it is an area of active research and the state of the art changes rapidly. Increasing network speeds are making the capacity of the NIDS platform a bottleneck that can compromise the effectiveness of the entire NIDS. Parallel NIDS platforms are a viable solution to address this problem.

This thesis discusses the unique requirements of a parallel network intrusion detection platform, describes a cost-effective yet scalable solution, and evaluates its performance using simulation. A custom NIDS load balancer distributes the processing load over an array of sensor nodes to minimize packet loss. It employs a scalable multi-level hashing technique to minimize NIDS vulnerabilities and to adjust to changing network traffic characteristics. The main contribution of this approach is the design of a load balancing technique that does not maintain per-connection state while still supporting stateful flow-based intrusion detection. Furthermore, the SPANIDS load balancer incorporates dynamic feedback from sensor nodes to adapt to changes in network traffic and processing load, further minimizing packet loss.

There is a price to be paid for the scalability benefits of using hash func-

tions to track network flows rather than a more stateful technique. When using only a hash to identify and route flows, the load balancer cannot know if a given adjustment to the routing table will break any flows over multiple sensor nodes. There is a high probability that such an adjustment will break flows, but no easy way to know how many. Therefore, there is an inherent tradeoff between connection breaks caused by adaptive routing adjustments and packet loss that could have been avoided with adaptive routing adjustments. The exact tradeoff depends on the characteristics of the network traffic being routed and the effectiveness of the adaptive adjustments in avoiding packet loss, but in general more adaptive adjustments mean more connections broken and fewer packets lost. The importance of connection breaks depends on the packet evaluation techniques used in the sensors and on the nature of the attacks that need to be detected. For example, a stateless NIDS sensor does not benefit from receiving a full flow. Additionally, in an overload attack, the contents of the connections are typically irrelevant to the attack, though such an attack may be used to mask a more detailed attack that would require analysis of the full flow. Thus, an exact optimal tradeoff cannot be determined in the general case.

Evaluation results demonstrate the performance potential of this approach. Since each adjustment potentially disrupts network flows, the best load balancing heuristic depends not only on the packet loss rates but also on the number of network flows must be broken to achieve this performance. An equally important consideration is the implementation complexity of different heuristics. Under these considerations, a heuristic that estimates the processing load that hash buckets exert on the sensor nodes outperforms all other

schemes. Additional parameters for this approach also have a noticeable impact on overall performance, allowing the fine-tuning of packet loss or broken connections to meet the needs of particular environments.

The popularity of the Internet has driven networks to increase dramatically in speed and size to satisfy the demands of the growing population. At the same time, the size and diversity of the Internet creates an irresistible plethora of motives and opportunities for malicious users to disrupt or otherwise abuse the resources connected to the Internet. For this reason, faster, cheaper, and better methods of protecting network users and network resources from remote mistreatment will always be in demand.

BIBLIOGRAPHY

1. A. V. Aho and M. J. Corasik. *Communications of the ACM*, 18(6), June 1975.

2. R. S. Boyer and J. S. Moore. *Communications of the ACM*, 20(10):762–772, October 1977. ISSN 0001-0782. doi: \url{http://doi.acm.org/10.1145/359842.359859}.

3. N. Brownlee and K. Claffy. Internet stream size distributions. In *Proceedings of ACM SIGCOMM*, pages 282–283, 2002. URL `citeseer.ist.psu.edu/brownlee02internet.html`. http://www.caida.org/analysis/workload/netramet/lifetimes/.

4. D. E. Comer. *Internetworking with TCP/IP*. Prentice-Hall, Inc., Upper Saddle River, fourth edition, 2000. ISBN 0-13-018380-6.

5. X. Corporation. Virtex ii datasheet, 2001. DS-031-1.

6. M. Crosbie, B. Dole, T. Ellis, I. Krsul, and E. Spafford. IDIOT - user guide. Technical Report TR-96-050, Purdue University, West Lafayette, IN, US, Sep 1996. URL `citeseer.ist.psu.edu/crosbie96idiot.html`.

7. O. P. Damani, P. E. Chung, Y. Huang, C. Kintala, and Y.-M. Wang. *Computer Networks and ISDN Systems*, 29(8-13):1019–1027, 1997. ISSN 0169-7552. doi: http://dx.doi.org/10.1016/S0169-7552(97)00030-5.

8. H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational experiences with high-volume network intrusion detection. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 2–11, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-961-6. doi: http://doi.acm.org/10.1145/1030083.1030086.

9. S. Edwards. Vulerabilities of Network Intrusion Detection Systems: Realizing and Overcoming the Risks. URL `http://www.toplayer.com/content/resource/white_papers.jsp,requiresregistration`. May 2002.

10. A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Symposium on Operating Systems Principles*, pages 78–91, 1997. URL `citeseer.csail.mit.edu/fox97clusterbased.html`.

11. C. Giovanni. Fun with packets: Designing a stick. Whitepaper, March 2001. URL `http://www.eurocompton.net/stick/papers/Peopledos.pdf`.

12. J. W. Haines, R. P. Lippmann, D. J. Fried, M. A. Zissman, E. Tran, and S. B. Boswell. 1999 darpa intrusion detection evaluation: Design and procedures. Technical Report 1062, Lincoln Laboratory, Massacheusetts Institute of Technology, Lexington, Massachusetts, February 2001.

13. S. N. S. Inc. Snort 2.0 - detection revisited, 2002.

14. C. S. Incorporated. Cisco 7500 gigabit ethernet interface processor (geip). whitepaper, 1998. http://www.cisco.com/warp/public/cc/pd/ifaa/ifpz/ggethifpz/prodlit/geip_ds.pdf.

15. A. N. S. Institute, editor. *IEEE Standard for Information Technology: Portable Operating System Interface (Posix)*. Institute of Electrical & Electronics Engineers, January 1994. ISBN 155937375X.

16. S. Iyer and N. McKeown. Making parallel packet switches practical. In *INFOCOM*, pages 1680–1687, 2001. URL `citeseer.ist.psu.edu/iyer01making.html`.

17. V. Jacobsen, C. Leres, and S. McCanne. tcpdump. available via anonymous ftp to ftp.ee.lbl.gov, June 1989.

18. C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful intrusion detection for high-speed networks. In *Proceedings of the IEEE Symposium on Research on Security and Privacy*, pages 285–294, Oakland, CA, May 2002. IEEE Press.

19. J. Larsen. Hogwash. http://hogwash.sourceforge.net.

20. K. Levchenko, R. Paturi, and G. Varghese. On the difficulty of scalably detecting network attacks. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 12–20, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-961-6. doi: http://doi.acm.org/10.1145/1030083.1030087.

21. R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. *Computer Networks: The international Journal of Computer and Telecommunications*

Networking, 34(4):579–595, 2000. ISSN 1389-1286. doi: http://dx.doi.org/10.1016/S1389-1286(00)00139-0.

22. MITRE. Spitfire intrusion detection environment user guide, August 2001. URL `http://downloads.openchannelsoftware.org/Spitfire/Handbook_PDF_v5.zip`.

23. B. Moore, T. Slabach, and L. Schaelicke. Profiling Interrupt Handler Performance through Kernel Instrumentation. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers & Processors*, San Jose, CA, October 2003. IEEE Press.

24. B. Mukherjee, L. T. Heberlein, and K. N. Levitt. *IEEE Network*, 8(3):26–41, May/June 1994. ISSN 0890-8044.

25. T. Nishimura and M. Matsumoto. A c-program for mt19937, with initialization improved 2002/1/26, January 2002. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/CODES/mt19937ar.c.

26. S. Northcutt. Shadow. Technical report, Naval Surface Warfare Center, Dahlgren Laboratory, 1998. URL `http://www.nswc.navy.mil/ISSEC/CID/`.

27. M. Norton. Optimizing pattern matching for intrusion detection. Technical report, SourceFire, Columbia, Maryland, September 2004.

28. V. Paxson. *Computer Networks*, 31(23–24):2435–2463, December 1999.

29. N. J. Puketza, K. Zhang, M. Chung, B. Mukherjee, and R. A. Olsson. *IEEE Transactions on Software Engineering*, 22(10):719–729, 1996. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.544350.

30. M. J. Ranum. Experiences benchmarking intrusion detection systems. http://www.itsecurity.com/papers/nfr2.htm, February 2002.

31. M. Roesch. Snort: Lightweight intrusion detection for networks. In *Usenix LISA '99 Conference*, Berkeley, California, 1999. Usenix Society.

32. L. Schaelicke, T. Slabach, B. Moore, and C. Freeland. Characterizing the Performance of Network Intrusion Detection Sensors. In *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, Lecture Notes in Computer Science, pages 155–172, Berlin - Heidelberg - New York, September 2003. Springer-Verlag.

33. T. Toth and C. Kruegel. Connection-history based anomaly detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2002. URL `citeseer.ist.psu.edu/toth02connectionhistory.html`.

34. H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 193–204, Portland, Oregon, USA, 2004. ACM Press. ISBN 1-58113-862-8. doi: http://doi.acm.org/10.1145/1015467.1015489.

35. S. Wu and U. Manber. *Communications of the ACM*, 35(10):83–91, 1992. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/135239.135244.

36. S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Technical Report TR-94-17, Department of Computer Science, University of Arizona, May 1993. URL `citeseer.ist.psu.edu/wu94fast.html`.

This document was prepared & typeset with pdfLaTeX*, and formatted with* NDdiss2$_\varepsilon$ *classfile (v3.0[2005/07/27]) provided by Sameer Vijay.*