Lehrstuhl für Netzwerkarchitekturen
Fakultät für Informatik
Technische Universität München

Bachelor-Arbeit

Transparent Load-Balancing for Network Intrusion Detection Systems

Matthias Vallentin

Aufgabensteller :		Univ.-Prof. Anja Feldmann, Ph.D.
		Intelligent Networks and Management of Distributed Systems,
		Deutsche Telekom Laboratories / TU-Berlin
Betreuer :	1.	Dr. Robin Sommer
		International Computer Science Institute, Berkeley, USA
	2.	Holger Dreger
		TU München
Abgabedatum :		15. November 2006

Eidesstattliche Erklärung

Ich versichere, dass ich diese Bachelor-Arbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. November 2006

Matthias Vallentin

Abstract

Since the amount of network traffic continuously increases, security analysis in large-scale environments faces new challenges to keep pace with the rapidly growing traffic volume. Network intrusion detection systems (NIDS) form an integral part to secure the network perimeter by steadily inspecting network traffic in order to detect security breaches. The traditional single-machine architecture of a NIDS cannot provide enough resources to cope with the growing traffic volume. Vendors offer expensive solutions based on custom hardware. Aimed at high-performance environments, these systems seem to sustain the induced load, however, they fall short in providing a cost-effective and flexible solution.

With our work, we set out to combine the performance of custom hardware with the flexibility and cost-efficiency of standard hardware. By distributing the network traffic stream over an expandable array of machines, we build a NIDS cluster suited for high-performance environments. In their entirety, the machines form a transparent NIDS cluster based on commodity hardware. To distribute the work load, each instance of the NIDS conducts analysis on a disjunct subset of the network traffic. A key challenge remains the exchange of lacking decision context. Facing this challenge, we discuss important design guidelines for NIDS clusters that promote the construction of effective implementations for practical use.

Furthermore, we thoroughly evaluate our cluster with respect to accuracy and performance. Having a reliable testbed in place, we perform various measurements which yield insightful results. Based on our observations, we draw the conclusion that our approach provides a viable solution for large-scale networks.

This thesis begins with a recapitulation of basic concepts of network intrusion detection. Particularly, we emphasize the main subject of our studies, the open-source NIDS Bro. After presenting its architecture and communication framework which our work inherently relies on, we highlight challenges that high-performance environments pose. Thereafter, we analyze objectives and mechanisms relevant for transparent load-balancing. Finally, we present our transparent load-balancing NIDS cluster, operating in a large-scale research network at the Lawrence Berkeley National Laboratory.

Zusammenfassung

Das ständig wachsende Datenvolumen stellt für die Sicherheitsanalyse in leistungsstarken Gbps-Netzwerken neue Herausforderungen dar. Network Intrusion Detection Systeme (NIDS) bilden dabei einen wesentlichen Bestandteil zur Sicherung des Netzwerkes, indem sie den Netzwerkverkehr hinsichtlich bösartiger Aktivitäten überwachen. In Umgebungen mit hohem Datenaufkommen haben bisherige Ansätze, deren Architekturen auf Einzelbetrieb ausgelegt sind, ihre Grenzen erreicht. Um der unzureichenden Rechenkapazität entgegen zu wirken, bieten Hersteller meist sehr teure, speziell zugeschnittene Hardware an. Abgesehen vom hohen Preis bieten diese Systeme nur unzureichende Flexibilität für die Dynamiken von Hochleistungsnetzwerken.

In dieser Arbeit stellen wir Methoden zum *Clustering* und *Load-Balancing* von NIDS auf Standard-Hardware vor, die wir am Beispiel des open-source NIDS Bro in die Praxis umsetzen. Wir versuchen, die Synergien aus der Kombination der Performance von Spezialhardware mit der Flexibilität und den Preisvorteilen von Standardhardware auszunutzen. In diesem Zusammenhang entwickeln wir einen NIDS-Cluster, indem wir den Netzwerkverkehr auf mehrere Maschinen verteilen, die jeweils eine disjunkte Teilmenge des Gesamtverkehrs analysieren. Kern unserer Untersuchungen stellt dabei die effiziente Realisierung des Austauschs von *Stateinformationen* dar, die den einzelnen NIDS-Instanzen bei ihrer Analyse fehlt.

Ferner führen wir eine intensive Evaluation unseres Clusters durch. Erkennungsgenauigkeit und Performanz stellen dabei wesentliche Aspekte unserer Messungen dar. Unsere Ergebnisse zeigen, dass unser Ansatz tatsächlich eine brauchbarere Lösung für große Gbps-Netzwerke bietet.

Zu Beginn unserer Arbeit stellen wir grundlegende Konzepte von Network Intrusion Detection dar. Insbesondere gehen wir konkret auf das Bro NIDS ein, das wir als Ausgangspunkt für unsere Arbeit verwenden. Nachdem wir die Architektur und das Kommunikations-Framework behandelt haben, widmen wir uns den Herausforderungen von Hochleistungsnetzwerken. Im Anschluss entwickeln wir Konzepte und Mechanismen für transparentes Load-Balancing, die wir mit unserem NIDS-Cluster umsetzen. Abschließend evaluieren wir unsere Implementierung. Unser NIDS-Cluster ist mittlerweile in die Netzwerk-Infrastruktur des Lawrence Berkeley National Laboratory integriert.

Contents

1 Introduction

1.1 Motivation

Network intrusion detection systems (NIDS) monitor the network traffic in order to detect malicious activities and deviations from the site's policy. The effectiveness of a NIDS is not only determined by the sophistication of the analysis but also by the available system resources. High-performance NIDSs face ambitious challenges to cope with the constant increase of data volume. Conventional architectures which are limited to operate on a single machine now exceed their resource limits in terms of CPU power and packet capturing performance. If the analysis cannot keep up the induced network load, the resulting packet drops significantly worsen the detection rate of NIDSs.

The volume of network traffic is continuously growing. To thwart scalability limitations, vendors provide closed-source solutions based on expensive custom hardware. Despite improved performance, their inflexibility and high costs leave them as a second choice.

Our work presents a load-balancing NIDS cluster on commodity hardware as a fruitful alternative which combines the performance of custom solutions with the flexibility and cost-efficiency of standard hardware. To distribute the work load across an expandable set of NIDS instances, the incoming network packet stream is divided into slices of manageable size, whereas each instance performs individual analysis on a disjunct subset of the entire traffic. Clearly, the price for this type of load-balancing is the lack of valuable decision context on each instance.

A key challenge is to distribute the processing over multiple instances while at the same time maintaining the accuracy and depth of analysis a single NIDS could in principal achieve. To this end, a NIDS usually employs a communication sub-system enabling individual instances to exchange information in order to augment their limited view with the lacking context.

In this thesis, we set out to build a transparent cluster for network intrusion detection systems, i.e. a NIDS that appears to be a single entity, but in fact is a set of nodes, each processing a subset of the entire work. While most NIDSs only correlate high-level information that is already coined with the site's policy, we tackle the problem one step lower: leveraging the flexible communication framework of the open-source NIDS Bro, we build a load-balancing cluster in which every node is equipped with the same fine-grained decision context.

Our target environment is the Lawrence Berkeley National Laboratory, where our cluster has now replaced the hitherto existing setup. Moreover, our cluster extensions to the Bro NIDS will be part of the official Bro distribution.

1.2 Outline

The remainder of the thesis is organized as follows.

Chapter 2. The second chapter gives an overview about basic concepts of network intrusion detection. We explicate necessary terminology and familiarize the reader with the topic. Further, we present the subject of our studies, the open-source NIDS Bro with its flexible communication framework which we take as starting point for our work. Before discussing related work at the end of this chapter, we highlight the challenges that NIDS face in high performance environments.

Chapter 3. In the third chapter, we introduce transparent load-balancing. After outlining objectives of transparent load-balancing, we turn to prolific mechanisms which can be employed in high-speed environments. The remainder of this chapter is devoted to our concrete realization of the outlined goals: introducing our Bro cluster, an array of expandable machines performing distributed analysis. Through the discussion we sketch practical hurdles we encountered and provide adequate solutions where possible.

Chapter 4. In the fourth chapter, we conduct the evaluation of the Bro cluster. To this end, we use a captured trace to instrument the accuracy and performance of our cluster. We start with an investigation of the accuracy and then turn to a detailed performance analysis.

Chapter 5. In the last chapter, we recapitulate our work and provide concluding remarks with an outlook to future work.

2 Network Intrusion Detection

This chapter presents the fundamentals of network security related to network intrusion detection. As we set out to construct a NIDS cluster, it is important to understand core concepts of network intrusion detection. Moreover, we pave the way for comprehending relevant aspects of load-balancing in the next chapter.

After introducing basic concepts of network security in §2.1, we give an overview of network intrusion detection in §2.2. In section §2.3, we introduce the open-source Bro NIDS which is the main subject of our studies. Thereafter, we present characteristics of high-performance environments in §2.4, followed by a discussion of related work in §2.5.

2.1 Network Security

Computer security and in particular network security is based on three pillars: *confidentiality*, *integrity*, and *availability* [Bis03]. The interpretation of these vary and depend on the given environment. Further, the customs of a particular organization tailor their meaning.

2.1.1 Policy and Mechanism

Since the definition of benign and malicious behavior may differ across different sites, the decision whether a security breach took place depends on the site's *security policy*. A security policy is a manifestation of "what is, and what is not, allowed" [Bis03]. Ideally, policies are formulated mathematically, as a set of allowed and disallowed states, to be as precise as possible. Yet policies are mostly presented in colloquial language which leads to ambiguity.

To enforce a security policy, a *security mechanism* provides a "method, tool, or procedure" [Bis03]. On the one hand, mechanisms can be nontechnical, for example requiring a proof of identity to continue with a particular action. On the other hand, mechanisms can be technical, such as applications controlling access to restricted areas or systems. In practice, security devices often immingle policy and mechanism. These *implicit* declarations of a policy are often hard to comprehend because the introduced policy may seem too diffuse and ambiguous. A clear separation from policy and mechanism is essential to achieve a flexible configuration of security devices [TS02].

2.1.2 Threat Model

Already the "potential" violation of security is considered as a *threat* [Bis03]. It is not necessary that a breach actually took place to refer to a threat. Rather, every possi-

ble action violating the security policy is considered as a *vulnerability*. An *attack* or *intrusion* is an action performed by an *attacker* or *intruder* trying to exploit a vulnerability [Som05].

To quantify the extent a breach of security dimensions implies, one has to understand what to defend against and what to protect. The impact of a successful intrusion depends on the involved severeness. For example, the disclosure of personal identity information or other sensitive data can entail embarrassing newspaper articles, causing loss of reputation with high monetary consequences. However, non-critical information leakage might not impose any consequences at all.

Therefore, one has to get a picture of the enemy. Who are the bad guys? What are their incentives?

Historically, the majority of network attacks were performed by individuals against a single well-known target. Vandals demonstrated their programming prowess to attain respect in the underground community. These *directed attacks* were mostly conducted *manually*, with a certain amount of interactivity needed in order to succeed. The continuous growth of the Internet leveraged the spread of manifold attack tools, enabling even non-skilled people to launch sophisticated attacks. Most of these toolkits include *automated* attack sequences, facilitating the application for *script kiddies* who lack the capability to perform the attacks manually.

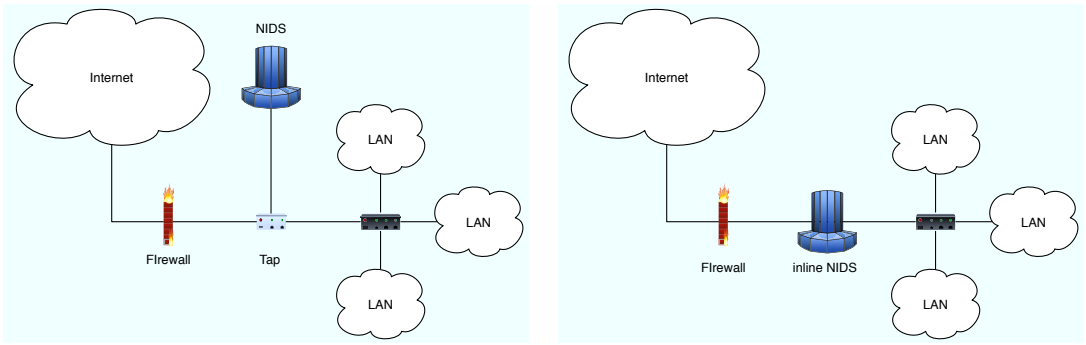
Nowadays, we observe a shift in motivation of malicious activity leading towards organized crime. This "cyber warfare" causes damage in a much higher order of magnitude. Crooks recently figured out how to make money with network attacks, for example by selling infected computers (also "zombies" or "bots") in order to perform DDOS attacks and sending spam. Their economic incentives fuel innovative approaches with an increasing degree of sophistication. Barford and Yegneswaran [BY06] set up the thesis that today's predominant *reactive* approaches to secure networks are not sufficient anymore, and more *proactive* methods are essential to get this problem under control.

Unsolicited *mal*icious soft*ware*, referred to as *malware*[1], that damages or infiltrates a computer has evolved continuously with big pace over the past years. Malware employs obfuscation techniques like *polymorphism* [KRVV04] and *metamorphism* [Szo05] to hide the original malicious intent of the code. But even software with benign intent, such as *Skype* [Sky], adopts similar techniques to escape from firewalls and systems responsible to surveille the network. This makes it an especially difficult and subtle affair to distinguish between malicious and benign.

Not only does the sophistication of attacks raise, but also the increasing volume, dynamics and complexity poses new challenges on the defending side. Particularly the high work load motivates us to create both robust and accurate network protection systems.

[1] A classification of malware is not in the scope of this thesis, yet a detailed terminology is explicated in [Szo05].

Figure 2.1 Different deployment schemes.



(a) Classical NIDS.

(b) Inline NIDS.

2.2 Network Intrusion Detection

The growing tendency of automated and undirected network attacks render manual forms of auditing ineffective. Therefore an automatic system, an *Intrusion Detection System* (IDS), aids in detecting intrusion.

In the following, we introduce required terminology orientated at [Som05]. Primary source of a *Network Intrusion Detection System (NIDS)* is network traffic. As opposed to a NIDS, the scope of a *Host Intrusion Detection System (HIDS)* is limited to the analysis of singular hosts. It is further possible to build *hybrid* systems, for example by enriching a NIDS decision context with host-specific information [Bro]. Our work resides at network level, hence we focus on NIDSs throughout the remainder of the thesis.

A NIDS raises an *alert* when it thinks that it detected a security breach. Alerts are either presented textual, embodied in *log* files, or displayed via a *graphical user interface (GUI)*. The notifications can then be dissected by a human analyst or an automatic post-processing facility. A correctly identified intrusion is termed as a *true positive*, whereas a reported false alarm manifests as a *false positive*. If the NIDS does not recognize an ongoing intrusion, we face a *false negative*. But if it correctly remains calm when no breach occurred, we encounter a *true negative*. It is important to note that these definitions depend on the site's policy and do not represent an absolute classification [Som05]. As an example, certain activity can yield a true positive on one site's policy, but may be legitimate behavior on other sites, thus yielding a false positive with an identically configured NIDS.

In a *passive* setup, the NIDS monitors traffic using a *network tap*, visualized in Figure 2.1(a). In contrast, traffic can also flow directly through a NIDS as depicted by Figure 2.1(b). Such a system is called an *in-line NIDS* or *Network Intrusion Prevention System (NIPS)*. All NIPS are by definition *active* as they can inspect every network packet and —if no malicious activity found— put it back on the wire. Even so a NIDS can be active and react on dangerous activities, e.g. dynamically block network traffic that it believes to be malicious.

Ideally, intrusions are detected and even blocked before they reach their target. Most of the commercial products today name themselves NIPS to advertise their additional capability of stopping intrusions. Moreover, the reputation of intrusion *detection* systems has decreased by reason of the disputed Gartner report [Sti03]. The report demonstrated the limitations and drawbacks of intrusion detection and the reasons why intrusion prevention is a fairly better method of securing a network. Vendors quickly adapted their product descriptions according to the report. Even non in-line systems have been rebranded to survive at the market. Due to the ambiguity of the term intrusion "prevention", we prefer the term *in-line NIDS* when referring to a system that follows the pristine intent of a NIPS.

2.2.1 Architecture

Conceptually, a NIDS is not a single monolithic box, but rather consists of modular components. In order to distribute instances of components in §3, we first have to understand what types of components exist. Since the discussion of components often implies aspects of the detection strategy [Bac00], a revised model has been introduced in [Som05]. This refined abstraction includes five types of components as shown in Figure 2.2:

Collector. Providing an interface to access one or more data sources, the collector aggregates data and sends it to the detection engine. Most NIDSs operate on a raw *packet stream*, usually forwarded to the collector from a *network tap*.

Detector. The very heart of the NIDS is the detector which performs detection. Feeded with the data provided by the collector and storage component, it generates alerts when it believes to disclose malicious activity.

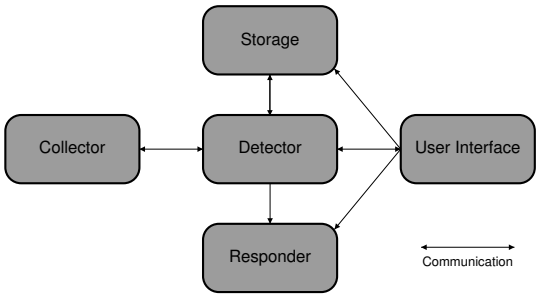
User Interface. The user interface is required for any form of interaction with the user. It reports alerts, provides a control interface, and enables the user to weave the site's policy into the NIDS.

Storage. Data such as suspicious activities or even successful intrusion can provide valuable insight for forensic analysis and qualify hence for persistent storage. Either the detector or the user interface can access the storage component. Frequently, a database system is employed to increase storage performance.

Responder. Revealed intrusions can trigger reactive responses in order to prevent subsequent attacks that further compromise the system. For example, connectivity dropping is an active countermeasure to dike further damage. Moreover, a response can also be generated manually through the user interface.

Especially in large-scale installations, components may be instantiated more than once and can be physically distributed across the site. However, all components are usually accommodated in one single piece of software. As we target high-speed environments, *replication* of particular components becomes inevitable to thwart performance

Figure 2.2 Components of a NIDS identified by [Som05].



degradation [TS02]. Separated components, however, need to communicate in order to exchange information. To this end, a modular NIDS has to feature a communication sub-system (see §3.3.2).

2.2.2 Detection strategies

A NIDS employs one or more *detection strategies*[2]. Many different strategies have been examined in literature so far, but according to [Som05], they can be narrowed down to *misuse detection*, *anomaly detection*, and *specification-based detection*. We sketch each of them as follows.

Misuse Detection. A system based on misuse detection recognizes events indicating *known* attacks that are looked up in a library of *attack patterns* (or *rule set* [Bis03]). When such a pattern successfully matches in the network stream, a potential intrusion is reported. Ideally, the supplied library of attack patterns can even detect previously unknown attacks by learning from the past. Since the attack pattern library is the system's heart, it has to be carefully maintained and constantly kept up to date.

Anomaly Detection. Anomaly detection systems seek for unexpected behavior that could be evidence of an intrusion. For this purpose, they are equipped with metrics of *expected* behavior. Any significant deviations from the metrics raise an alert. In practice, anomaly detection alone is very prone to false positives [Som05], but has reported to work successfully in well-defined application domains [KV03, KMVV03].

Specification-based Detection. In contrast to misuse detection, specification-based systems define explicitly *allowed* behavior. Other observed behavior than the specified is interpreted as a violation of the site's policy. Given that specification-based detection is the inverse of misuse detection, both approaches are equal in terms of their expressiveness. In operational environments, it is rather impractical to specify a *complete* set of benign events. Nevertheless, there are some practical applications specifying allowed communication behavior. As an example, firewall

[2] Bishop refers to them as *models* [Bis03].

rules [CBR03] attest which hosts are allowed to establish connections to other hosts.

2.2.3 State

In order to make a decision, a NIDS needs to rest on *decision context*. The more context a NIDS can incorporate, the more reliable it can conduct network analysis. There are various sources to gather decision context from. For example, one can feed the NIDS with information about the network topology. Yet the most context is accumulated by the NIDS itself during runtime, as its own dynamic view of the network. This view of the current communication in the network is referred to as *state*. State continuously changes and evolves over time as the communication in the network does.

A *stateful NIDS* incorporates various types of state, whereas a *stateless* NIDS regards each packet on its own as a self-contained unit of analysis. Historically, a clear distinction between stateful and stateless network intrusion detection could be made. Nowadays, all major NIDS operate stateful, yet differ in the magnitude of state they accommodate.

Our work is inherently based on the *exchange* of state. We construct a transparent NIDS cluster in §3, where each node performs analysis on a subset of the entire network traffic and thus holds only a fraction of the total available state. To get picture of the entire network communication, each node shares its own isolated perception with the other nodes.

We now present a few common types of state [Som05]. One type is *connection state*. Thereby, meta information of every active connection is stored, for example the address of the connection origin and destination, the duration, payload volume, and current handshake status. It is also possible to treat state as *per-host state* by storing all connection attempts for a given originator address. In §3.4.3, we examine this type of state as part of a scan detector in detail. Further, *signature state* embodies the current progress of a signature match. Because of their high expressiveness, many NIDS allow the representation of signatures in regular expressions. However, matching hundreds of signatures simultaneously can potentially lead to state explosion due to the exponential growth of the underlying DFA [HU79].

In addition to the state gathered by itself, a NIDS can be supplied with external context to include host-based analysis [Bro]. Valuable information that cannot be detected on the network, such as operating system internals or decrypted data, greatly enhance the decision context of a NIDS.

2.2.4 Trade-Offs and Limitations

Network intrusion detection systems are confronted with fundamental trade-offs and limitations. In the following, we recapitulate inherent basic constraints that affect the deployment of every NIDS in high-performance environments [Som05]. We also sketch how these constraints can impact a NIDS cluster.

Effectively deploying a NIDS is expensive. Not only the initial setup generates costs, but constant updates for the system and maintenance have to be taken into account,

too. Moreover, analysts need a thorough understanding of the system's output in order to advise appropriate actions. Often, most of these implicit costs are forgotten, consequently reducing the NIDS's benefit and value. Note that security is not a buyable product, but rather a continuous process. In terms of a NIDS cluster, cost-effectiveness can be achieved by employing standard hardware instead of expensive custom hardware. On the other hand, the higher administrative complexity a cluster imposes has also to be respected.

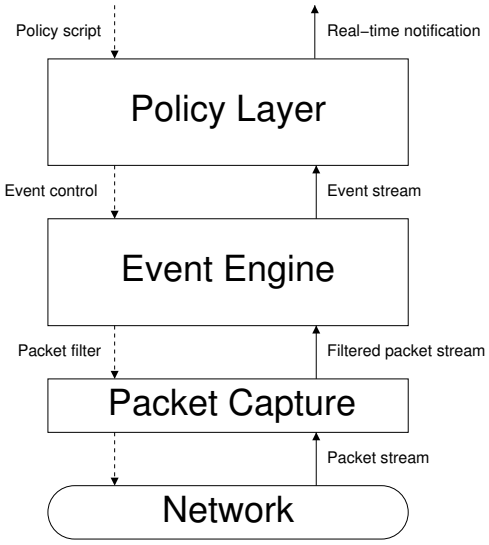
A fundamental limitation of NIDSs pose their *false positive rate*, that is, the proportion of false positives to true positives. Practice shows that the immense volume of benign data input too often yields too many false positives, thereby flooding the analyst with irrelevant output. Even worse, concealing intrusions by deliberately generating false positives exhibits an attack vector at the system itself, rendering it ineffective. The cause of a high false positive rate is usually a mismatch between the configured NIDS policy and the site's policy. As outlined in §2.1.1, the site's policy is commonly presented informally, lacking mathematical precision which complicates the integration into the NIDS. The resulting ambiguities implicate a high number of false positives. In addition, not many NIDSs offer a flexible configuration interface to incorporate a detailed mathematical representations of the site's policy. In context of a NIDS cluster, state propagation can significantly impair the detection rate. Due to missing or delayed arrival of decision context, a cluster node could make a wrong decision that would not have been made with full decision context. It is hence important to equip each node with the required amount of state and reduce state propagation latencies to a minimum.

The vulnerability of the NIDS itself imposes further limitations. From the attacker's perspective, evading the NIDS by delusion or subterfuge is an attractive means to conceal malicious activities and remain undetected. Various ways to bypass a NIDS have been debated in [PN98, HKP01, SP03]. Another attack vector form *denial-of-service (DoS)* attacks. Every NIDS faces hardware constraints in terms of CPU and memory. If an attacker manages to induce a very high resource consumption on the system, it will eventually crash or at least miss a subset of activities. Therefore, attack resilience is an important factor which has to be addressed in the early design phase of a NIDS. When devising a NIDS cluster architecture, it is import that the architecture itself does not expose any new exploitable attack vectors. For example, if one cluster node can be easily overflowed, the entire system is rendered ineffective.

2.3 Bro System

In the following, we introduce the NIDS we use throughout our studies. Not only the flexible communication sub-system that we present in §2.3.2 motivates our choice, but also the direct contact to the developers allows us to quickly report bugs and gain insight of the system's internals.

Figure 2.3 Architecture of the Bro [Som05].



2.3.1 Architecture

Bro [Pax99] is a very flexible open-source NIDS designed to operate in high-speed and large-volume environments. The author of Bro is Vern Paxson, still primarily involved in the development. As opposed to many other NIDSs, it is not limited to pursue only one detection strategy. The implementation of a specific strategy is independent of Bro's *policy-neutral* core. Major design principles are *(i)* high-speed, large volume monitoring, *(ii)* no packet filter drops, *(iii)* mechanism separate from policy, and *(iv)* resistance against evasion attacks, directed at the monitor itself.

To achieve these goals, Bro's system structure is divided into layers, as demonstrated in Figure 2.3. The lowest layer contains the most amount of data. When going higher up through the layers, the stream of data declines. A static *BPF* expression [MJ93] pre-filters the raw packets incoming the network interface. Leveraging *libpcap* [Lib], Bro remains portable running on various Unix flavors. Another advantage libpcap entails is that it reduces the traffic in the kernel. By applying a BPF expression, packets are discarded even *before* the kernel hands them up to the user-space, thereby substantially reducing the system load.

The filtered packet stream is passed to the *event engine* and generates *policy-neutral events*. These events do not imply any embossing with the site's policy (see §2.1.1). In fact, they represent "abstractions of network activity at different semantic levels" [Som05]. Example events are TCP connection established, HTTP request, FTP data connection expected, or POP3 login rejected.

Upon completion of the event engine's processing, the event stream is handed up to Bro's *policy layer*. Using a specialized high-level scripting language, called *Bro*, the user can weave the site's policy into the Bro NIDS. The scripts hold *event handlers* that are

executed when the corresponding event is generated. The code inside the event handlers can execute arbitrary commands of Bro's scripting language, e.g. raise alerts, log real-time notifications to files, create new events, or even start external programs to counter identified malicious activity.

In contrast to other NIDSs, Bro uses a connection (or *flow*) as main unit of analysis. A TCP connection consists of the source and destination address as well as the originating and destination port. Other protocols, such as UDP or ICMP, also possess a notion of a flow.

Recently, Bro features the *independent state* [SP05] framework for distributed analysis. It allows to share the hitherto volatile in-memory state among several instances of the system. As we fundamentally rely on independent state in our work, we now give an overview about its internals.

2.3.2 Independent State

Network Intrusion Detection Systems generally rely on managing a great amount of state (see §2.2.3). This state represents the NIDS's current view of the network communication. Unfortunately, the state resides in the volatile memory, exclusively accessible to a single process on a single host.

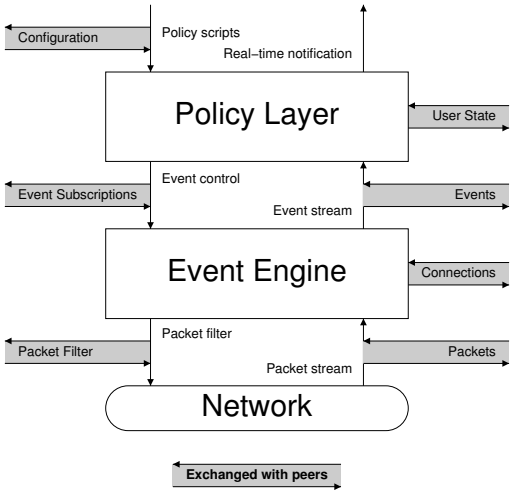
As the volume in high-speed environments raises, the memory consumption of the state increases in conjunction. Particularly the TCP connection states grow very fast. It is therefore necessary to either reduce the amount of consumed memory, e.g. by expunging state aggressively and improving state expiration heuristics, or to distribute the analysis across multiple instances in order to keep more state in total [Som05]. If the work load is split across n instances, ideally we have n times more resources. With this form of *load-balancing*, we can perform deeper inspection. This is exactly what we aim for: a load-balancing NIDS cluster thwarting scalability limitations by distributing the analysis over an expandable set of nodes.

Independent state [SP05] allows us to share the internal, fine-grained state of a NIDS among multiple instances. First, there is *spatially independent* state that can be transfered from one instance to another concurrently running instance. Second *temporally independent* state subsists even after the executing instance has exited. It can thereby be shared by subsequently running processes.

With independent state, a plethora of new applications is possible, including *(i)* selectively preserving key state across restarts and crashes, *(ii)* dynamic reconfiguration of the NIDS on-the-fly, *(iii)* user-level state tracking over time to support high-level policy maintenance, and *(iv)* detailed profiling and debugging [Som05].

Still the most interesting application independent state offers us is the flexible ways of distributed load-balancing. Since analyzing a high-volume traffic stream is very difficult for a single NIDS, distributed analysis opens a new dimension to mitigate this problem. In order to manage the high volume, we spread the analysis across several instances, each processing a disjunct subset of the entire traffic. Without any state sharing, valuable context in the form of state is missing to the other instances. The framework provided by independent state is able to disclose the state and make it available to other parallel

Figure 2.4 Independent state integrated into Bro [Som05].



running instances. Every instance still analyzes only a subset, but has the full decision context. The goal hereby is to maintain the same depth of analysis one single instance could principally achieve. We delve into this topic again in chapter §3.

The conception of state propagation is realized with a *serialization* framework. All of Bro's state can be converted in a self-contained serialized binary representation. Figure 2.4 depicts a breakdown of the different types of state that Bro exchanges. As illustrated, two communicating Bro instances, in the following termed as *peers*, can share core events, user-level events and user-level data.

2.4 High-Performance Environments

High-performance environments pose new challenges to the design and operation of a NIDS. We put out the differences in contrast to traditional environments, summarizing some key aspects relevant for network intrusion detection that have been intensively discussed in [Som05]. Note that all these observations apply to large and open research environments and not to private closed networks.

Policy. The site policy of most high-performance environments is derived from "terms of use" (TOU) that users accept when accessing the network. Generally, the TOU appear to be very liberal in research environments, because the network is an important tool for researchers. The intended use of the network is first enforced by firewall rules and second surveilled by monitoring devices. Due to the liberal and informally formulated constraints, the applicability of NIDS is limited. Users not complying with the TOU are often detected by the consequences their attack implies. For example, traffic patterns may change after setting up an illegal FTP server. An inferred policy from implicit consequences of attack is termed as

experience-based policy. This type of detection focuses primarily on internal hosts, since *(i)* the aim is to protect internal hosts, *(ii)* it is possible to contact a local administrator in charge, and *(iii)* tracking down external victims is impracticable, as contacting the responsible persons in charge turns out to be very cumbersome and often not effective.

Threats. Due to their size, large-scale environments face mostly *undirected* attacks. Their incentive is to concentrate on big attacks. But also singular intrusions are sought to be tracked down, since they may compromise critical selected targets. Although individual intrusions pose a certain risk, they do not vitiate the network's operation. Defending only against the "big fish" is a very pragmatic, but practicable approach.

Undirected attacks imply two main threats: *misuse of resources* and *worms*. While the former represents all kinds of malicious activities exploiting the available resources, the latter has the potential to quickly infect a huge number of hosts within minutes [SMPW04] and can thus harm the entire network. As public networks are constantly suffering from attacks [PYB+04], particularly high-performance environments are fruitful targets.

Traffic. As opposed to small environment, large networks exhibit a very divers traffic pattern. The network's *application-mix*, i.e., the "fractions that the major network applications contribute to the total volume" [Som05], has a great effect on the performance of NIDS. This is due to the fact that each application protocol is analyzed to a different depth. Over time, the traffic follows strong time-of-day and time-of-week effects. Furthermore, the prevalence of "heavy-tailed" data transfers [WTSW97, FGW98] can constantly spawn sudden peaks of traffic volume. Thus, in high-performance environments a NIDS needs to robustly handle not only the average case, but also the aforementioned frequently occurring bursts.

2.5 Related Work

The steadily increasing network volume motivates to focus on network intrusion detection systems designed for high-performance environments. We leverage Bro's independent state frame-work to build a load-balancing NIDS cluster. Unlike other approaches that correlate only high-level information such as logs or alerts, our approach is one step lower: we equip every cluster node with the same policy-neutral state. At the same time, each node analyzes only a subset of the entire traffic. Thus far, we have not encountered a similar approach in the research community.

The idea to employ clusters for scalable network services, however, is not new. Fox et. al mention several advantages clusters provide, including *incremental scalability*, *high availability*, and the *cost-effectiveness* of commodity PCs [FGC+97]. Although NIDSs are usually passive components, the identified advantages are also beneficial for such systems.

The performance of network intrusion detection has been extensively studied in the past [PZC+96, SSMF03, SWF05, SF05]. All studies conclude that it is imperative to cope with the induced load that the growing network traffic imposes. Schaelicke and Freeland argue that system-level optimizations such as *interrupt coalescencing* and *rule-set pruning* as well as architectural techniques can significantly improve performance and reduce packet loss [SF05].

While previous work primarily focuses on the design of a NIDS cluster processing front-end [SWF05, KVVK02], we look furthermore into the challenges that intra-NIDS communication reveals.

Numerous different NIDSs are available today. The focus and range of application vary for each system. To our knowledge, only a few systems feature a tunable and flexible communication sub-system that we can leverage to build a NIDS cluster.

Snort [Roe99] is the most common and widespread open-source NIDS. The original author of Snort is Martin Roesch, whose company *Sourcefire Inc.* now continues its development and sells Snort-based appliances. Snort also runs on commodity hardware and utilizes *libpcap* to enable platform independent packet capturing. The detection engine is misused-based (see §2.2.2). Around a core of numerous signatures, various plug-ins enhance its functionality. Contrary to Bro, Snort doesn't feature a communication framework enabling parallel processing. Alerts generated by multiple instances have to be aggregated manually in order to gain an omniscient view of the network's security condition. Despite the lack of a communication sub-system, Kruegel et. al built a flow-based load-balancer on top of Snort [KVVK02]. Unfortunately, this approach maintains connection tables to forward packets belonging to the same flow to the corresponding sensor. Further, it does not support inter-sensor communication.

The *State Transition Analysis Technique (STAT)* tool suite [VEK00] is a set of distributed intrusion detection tools developed by the Reliable Software Group at UCSB. STAT uses a misuse-based detection strategy that understands intrusions as sequences of attack scenarios modeled in a state transition diagram. It supports inclusion of network-based, host-based, and application-based sensors. Thus, it is a *hybrid* intrusion detection system (see §2.2). The *MetaSTAT Infrastructure* [VKB01] provides the communication sub-system and control infrastructure to enable distributed coordination of STAT-based applications. STAT-based tools fan out into $\{U,N,Net,Win,Web,Alert\}STAT$, each designed for a different application domain. In particular, *NetSTAT* [VK99] is the network-based component responsible for network communication. If it is impossible for the system to detect an attack completely, the partially configured scenario containing state information is propagated to other probes.

EMERALD [PN97] is highly-distributed hybrid intrusion detection framework developed by *SRI International*. It is designed to operate at large-scale enterprise networks and is not freely available. The architecture of EMERALD uses a layered approach to support hierarchical organization of monitors. Each monitor can subscribe to events and propagate correlated results.

Prelude [BOG03] is distributed NIDS that relies on the *Intrusion Detection Exchange Format (IDMEF)* [IDM] standard to exchange events. Several sensors are connected to managers which process and correlate alerts. In a distributed setup, multiple managers

can also act as relay managers that report to a central manager.

However, none of the existing approaches provided flexible enough means to share arbitrary policy-neutral state. In the following chapter, we present our approach leveraging low-level state propagation in order to create a transparent NIDS cluster.

3 Transparent Load-Balancing

In this chapter we present our approach how to effectively conduct network intrusion detection in high-performance environments. We introduce our motivation in §3.1. In §3.2, we present objectives of transparent load-balancing. To achieve these goals, we outline generic mechanisms in §3.3. We turn in section §3.4 to the minutiae of our work which includes a description of our target environment, the details of our enhancements of the open-source NIDS Bro, and a summary of our work.

3.1 Motivation

Our motivation is to build a transparent NIDS cluster, i.e. a load-balancing cluster of NIDS instances that appears to be a single entity, but in fact is a set of nodes, each processing a subset of the entire traffic.

As described in the previous chapter, NIDSs face various challenges that high-performance environments impose. The traditional single-machine architecture of a NIDS cannot provide enough resources to cope with the growing traffic volume. Vendors provide solutions on expensive custom hardware to tackle the new requirements. However, closed-source solutions are inflexible and too costly.

To bypass resource limitations, a common practice of many sites to date is to analyze only a fraction of the network traffic. The Lawrence Berkeley National Laboratory (LBNL, [LBL]), for example, filters out HTTP traffic on their main NIDS instance, because else the NIDS would not be able to cope with the induced load. An additional second NIDS instance is then responsible for dedicated HTTP analysis. Contrary to this approach, we want to analyze *all* network traffic in order to conduct effective and highly flexible network intrusion detection.

An alternative approach would be to employ *commercial off-the-shelf (COTS)* hardware and distribute the work load across an expandable set of NIDS instances, whereas each instance performs individual analysis on a disjunct subset of the traffic. While the available resources multiply with such an approach, each instance lacks valuable decision context (see §2.2.3). The NIDS instance itself has then to take responsibility for propagating state information to other nodes to reconstruct the whole picture.

Unfortunately, most of today's NIDSs lack the capability to restore the entire picture. They cannot share the internal fine-grained state which remains volatile in the memory of the NIDS. Once the NIDS terminates, this state is lost. Often, only high-level state (state that has already been coined with the site policy such as logs, alerts, etc.), is exchanged. To overcome this limitation, the *independent state* framework (see §2.3.2) has been developed, allowing us to propagate the bulky fine-grained state between instances

of a NIDS. Independent state offers manifold applications, yet we focus in this thesis only on distributed load-balancing.

While many NIDSs correlate only high-level information in a distributed NIDS setup, our approach is one step lower: we leverage independent state to build a cluster in which each node is equipped with the same decision context. Thereby, every node can rely on the same *policy-neutral* decision context even though it analyzes only a subset of the entire traffic. At the same time, we maintain the depth of analysis a single NIDS could principally achieve without resource limitations.

Having independent state in place, we face various other challenges on our way to build a transparent NIDS cluster which we now examine step by step throughout this chapter. In the following section, we discuss principle objectives that should be aimed at when conducting distributed network intrusion detection.

3.2 Objectives

In this section, we discuss goals of transparent load balancing. These goals should be interpreted as theoretical design guidelines when building a distributed NIDS cluster. Practical limitations often impede one or more of these goals, as we show in §3.3.

3.2.1 Transparency

An important goal of distributed systems is transparency. A load-balancing NIDS appears as *transparent* when we have the impression that we interact only with a single NIDS. Clearly, transparency is only perceived from the user's point of view. Behind the scenes, the NIDS aggregates the relevant information from multiple sources. It hides that its resources are distributed across several instances.

We need transparency since the characteristics of high-performance environments (see §2.4) force us to rethink network intrusion detection in a new dimension. A NIDS deployed in traditional environments would likely be overstrained if integrated in a large-scale network. In the event that it could cope with the load, it would still fall short in providing a tractable system. Conducting high-speed network intrusion detection requires a certain degree of transparency in order to achieve tractability.

In literature, transparency fans out into various aspects, as shown in Table 3.1. While distributed network intrusion detection in wide-area networks has to deal with a multitude of these aspects, load-balancing NIDSs involve mainly *replication transparency*, since we want to interact with a single system, and *concurrency transparency*, as the user usually does not notice which data structures are shared among the NIDS instances.

Beside the enumerated aspects of transparency, we identified two other important facets of transparency that are in particular relevant for load-balancing: *user interface transparency* and *accuracy transparency*. After discussing these two aspects, we argue why the remaining facets of transparency shown in Table 3.1 do not apply for load-balancing NIDS in general.

A deployed NIDS requires continuous attention to examine alerts and tune system parameters for maximum benefit. The user interface thus plays an important role in

Table 3.1 Several aspects of transparency of a distributed system [TS02].

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed.
Location	Hide where a resource is located.
Migration	Hide that a resource may move to another location.
Relocation	Hide that a resource may be moved to another location while in use.
Replication	Hide that a resource is replicated.
Concurrency	Hide that a resource may be shared by several competitive users.
Failure	Hide the failure and recovery of a resource.
Persistence	Hide whether a resource is in memory or on disk.

terms of the system's effectiveness [Som05]. There are many kinds of user interfaces for a NIDS, ranging from ASCII files to sophisticated graphical interfaces [ACI, Kre05, HS01, KO04]. The importance of transparency grows with the complexity of the environment. The best distributed NIDS is no advantage if the analyst cannot cope with its output. Therefore, it is crucial to provide a transparent user interface, allowing the user to interact with multiple instances of the system through a singular channel. At the same time, the user interface needs the flexibility to access the profound details of each node. It is a tightrope walk to catch the optimal degree of transparency while concealing irrelevant properties of the system.

Transparency can further be extended to the accuracy of a NIDS. The accuracy is represented by the *detection rate* of intrusions, meaning the ratio between *true positives* and actual intrusions [Som05]. Given that nodes have to exchange information to coordinate, the detection rate is affected by the underlying communication framework. For example, it is directly influenced by the speed messages need to travel to other nodes. If state updates do not arrive fast enough, valuable decision context might lack and hence impair the detection rate. Thus, a system whose detection rate does not pejorate in a distributed installation is said to be transparent with respect to its accuracy. As detecting intrusions is the very purpose of a NIDS, this aspect must be a priori an explicit design goal.

User interface transparency and accuracy transparency are explicitly wanted, whereas common interpretations of transparency generally do not hold for NIDS clusters. For instance, *Access transparency* is not the primary concern of a load-balancing NIDS, since byte order differences are not expected on cluster nodes because they usually employ the same hardware. If cluster nodes lie next to each other, *location* and *migration*, and *relocation transparency* are negligible aspects. From the operators point of view, *failure transparency* accompanies the usability of the system, e.g. when a backup node takes over the role of a defective node, thereby masking the failure. For load-balancing NIDSs, *persistence transparency* is explicitly unwanted, because it is important to know whether state information is already recorded to disk or still resides in the volatile memory.

It may further be tempting to blindly capsulate as much as possible from the user. Sometimes, this is not a good idea. An example is the component which is responsible for reporting system errors. We really would like to know where the error occurred to

take measures. Further, a high degree of transparency can negatively affect performance, as concealment and aggregation consume additional resources.

3.2.2 Scalability

Large-scale networks exhibit very dynamic properties and pose new challenges for network intrusion detection. In order to conduct effective network intrusion detection, we set out to build a *cluster* of NIDS instances. One fundamental advantage of clusters is their *scalability*: when the load offered to the system increases, an *incremental* and *linear* increase in hardware can maintain the same per-user level of service [FGC+97]. Here, incremental means that the system has the ability to grow incrementally over time, which is a major advantage as capacity planning for large-scale environments depends on high number of unknown variables. Linear in this context means that the amount of additional resources necessary is a linear function of the increase in offered load.

In literature, three different dimensions of scalability are differentiated [TS02]. First, if we can easily add more resources upon discovering the existent resources are not sufficient, the system is scalable with respect to its *size*. Secondly, if the resources may lie far apart from each other, a system is said to be *geographically* scalable. And thirdly, systems that are easy to manage even if they span many independent organizations are *administratively* scalable.

Consider scaling with regard to size. If a NIDS exceeds its physical resource limits, extending a single-architecture platform might be possible to some extent, but will eventually reach its physical constraints and be unable to cope with the load. To overcome this limitation, monolithic approaches have to be abandoned in favor of clustered and distributed architectures, not prohibiting further growth. Using cost-effective commodity hardware yields an optimal cost/performance ratio. At the same time, the inherent redundancy of clusters can be exploited to meet high availability and *failure transparency* (see Table 3.1).

While geographical scalability introduces demanding challenges for distributed network intrusion detection in wide-area networks, it can be neglected in the application domain of load-balancing. In this case, resources lie next to each other and are not confronted with the dynamics of wide-area communication.

Finally, a growing NIDS has to remain tractable and offer administrative scale. Not only have the system's internals an effect on administrative scalability, but also the embedding of the system in its environment. Scaling NIDSs across multiple administratively independent domains is a sophisticated task, involving many unsolved problems. For instance, conflicting site policies can significantly hinder the applicability of a NIDS.

In this section, we framed objectives for load-balancing in network intrusion detection. We identified transparency as an important goal to render distributed NIDSs tractable in large-scale networks. Transparency includes various aspects, such as the user interface and the accuracy of the NIDS. Another basic objective is scalability. Systems can scale with respect to their size, their geographic expansion, and their administrative complexity.

Table 3.2 Classification of traffic division schemes.

Scheme	Description
Static	Match a fixed criterion.
Dynamic	Match a varying criterion dependent on the incoming traffic.
Stateful	Distribution relies on accumulated information.
Stateless	Distribution does not require stored information.
Packet-based	Consider a packet as unit of distribution.
Flow-based	Consider a flow as unit of distribution.
Fair	The division schemes achieves an equal distribution among all nodes.
Unfair	The division schemes exhibits non-uniform distribution characteristics.

3.3 Mechanisms

This section presents prolific mechanisms of transparent load-balancing enabling us to conduct effective analysis in practice. Thereby, we address the previously emphasized design goals to come up to a viable solution for high-performance environments.

3.3.1 Load-Balancing

If a NIDS cannot process the incoming packet stream at full wire speed, buffers will eventually lead to packet loss or at worst overflow and potentially crash the system. Because packet loss can significantly worsen the attack detection, it must be avoided in any case.

To this end, we employ *load-balancing* to distribute the vast load equally among several *nodes*. On each node, an instance of a NIDS is performing analysis. The connected instances of the NIDS form a *NIDS cluster*. Ideally, the available resources multiply in such a cluster setup, meaning that we can toss in new nodes as soon as we realize that existent resources are insufficient.

Subject of load-balancing is the incoming network packet stream which is partitioned according to a *division scheme*. In general, division schemes may exhibit *static* or *dynamic* properties. The static scheme matches a fixed criterion, whereas a dynamic scheme uses a varying criterion that depends on the incoming traffic. Division schemes can also be *stateful* or *stateless*. A stateful scheme relies on stored information to decide to which node an incoming packet has to be forwarded. Stateless division on the other hand does not accumulate any kind of state. Further, we classify division schemes as *packet-based* or *flow-based*. Packet-based approaches consider a single packet as unit to distribute, whereas flow-based schemes consider a connection as distribution unit. At the same time, overloading any of the instances would introduce a new vulnerability. Hence, a traffic division scheme can either be *fair* or *unfair*. Table 3.2 summarizes the different division schemes once again.

All of the division schemes have one thing in common: correlated information between different partitions have to be exchanged by the NIDS instances. The load-balancer is only responsible for an efficient distribution, yet the NIDS itself has to provide a

communication sub-system in order to send the lacking information to other instances of the system.

Generally, schemes can be combined in large-scale environments to tune a NIDS cluster for maximum performance and efficiency. We briefly rate the different division schemes in Table 3.3. A scheme is *(i) fair* if it distributes the traffic equally over a set of nodes, *(ii) scalable* if adding further nodes is possible without negative effect on the load-balancer, and *(iii) flexible* if it does not impose limitations on the NIDS cluster. We have identified four relevant traffic division schemes whose advantages and disadvantages we discuss in the follow. In addition, we argue why only one is suitable for our needs.

Round-Robin Division. The easiest traffic distribution scheme can be achieved with round-robin division. In this case, packets or flows are evenly scattered over multiple instances. Round-robin division has the advantage that it is very easy to implement and provides a very fair load distribution. A round-robin distribution scheme based on packets is mentioned in [SWF05]. Its main disadvantage is that it disrupts flow information, rendering it unusable and inefficient in practice since today's NIDSs perform connection-based analysis. Packet-based division schemes impose such an immense communication overhead that it would be almost the same to sending the entire packets from instance to instance instead of only exchanging more light-weight information. Hence we discard packet-based schemes throughout the rest of our discussion.

If we use flow-based round-robin division, we could tackle this problem. However, in this case the load-balancer has to store state information for each flow in order to send the corresponding packet to the correct node. Despite the fairness this approach implies, we consider it as an impractical division scheme because it does not scale.

Dividing by IP Space. A balanced division by local IP space [Som05] requires knowledge of network traffic characteristics. These characteristics can be gained by performing traffic measurements and leveraging the administrator's operational experience. The main advantage this scheme offers is the simple integration of additional systems to further distribute the load. Further, intra-subnet communication does not involve any NIDS communication overhead as this type of communication is treated by the same NIDS instance.

On the downside, no inter-subnet activities (e.g. scans) can be correlated without communication. Moreover, since this approach is static, it cannot quickly adapt to changing traffic patterns.

Dividing by Application. Load division by application [Som05] delegates applications that form a significant share of the load to dedicated systems. For an example, if we find that the network traffic includes a large proportion of HTTP traffic, excluding HTTP processing from the main system and moving its analysis to a dedicated machine can effectively absorb protocol-specific peaks. But not every protocol

Table 3.3 Rating of the different division schemes.

	Round-Robin[a]	IP-Space	Application	Hash-based
Fairness	++/++	−	−−	++
Scalability	+/−−	++	−	++
Flexibility	−/++	+	−	++

[a]The Round-Robin scheme is examined for packet-based and flow-based division.

communicates within a self-contained single connection. Consider the FTP protocol whose control connection negotiates a subsequent data connection. Unless the control connection is successfully parsed, the corresponding FTP data connection cannot be recognized as such.

Yet the biggest problem is the poor scalability and unfair division of the traffic. Eventually, all applications that comprise a significant portion of the traffic will be assigned to particular nodes.

Hash-based Division. We differentiate between division by *classical flows* and *IP flows*. Classical flows are usually identified by a connection 4-tuple. For TCP and UDP, a flow is composed of all packets belonging to the same connection with identical originating IP, destination IP, source port, and destination port. Other transport protocols may exhibit similar flow-like definitions. The main advantage of this scheme is that each NIDS instance can perform connection-based analysis. In addition, it enables a scalable topology independent processing [KVVK02, SWF05]. Yet there are still cases where additional context is needed, for example the aforementioned FTP data problem is not solved in this scheme.

Dividing by IP flows would strike the FTP data problem, as control connection and data connection own the same IP source and destination pair, and the corresponding packets would be routed to the same instance. However, IP flow division is not as fine-grained and fair as classical flow division. For example, all connections belonging to a *vertical scan*[1], i.e. a port scan of a host, would fall to the same instance. This introduces a new vulnerability to the NIDS, as one instance can easily be overloaded. During such a denial-of-service attack, the system is less likely to detect actual attacks.

We favor hash-based division with classical flows since it yields the same fairness as flow-based round-robin division, but features much greater scalability due to its stateless property. Throughout the remainder of the thesis, we hence employ this division scheme.

Note also that load-balancing takes place independently from the actual NIDS technology. In principle, any NIDS can be accommodated in a cluster installation. However, the NIDS should feature a communication sub-system to exchange the lacking information.

[1]As opposed to a vertical scan, a *horizontal scan* probes the same service at multiple machines. It is also referred to as an *address scan*.

We examined various schemes to divide the network packet stream into subsets of manageable size. The crucial trade-off is how to divide the traffic without tearing apart information necessary to detect attacks and at the same time maintaining a fair load distribution. Apparently, it is impossible to address both ends satisfyingly. In the following, we therefore introduce communication mechanisms to mitigate the lack of information caused by the traffic division.

3.3.2 Communication

Distributed processing is key to fruitful load-balancing, as a single NIDS cannot cope with the load that high-volume links induce. Therefore, it presents a viable alternative for high-speed environments. However, a key challenge remains the coordination of their operation.

Today's NIDSs operate in real-time. A fast and flexible communication sub-system is essential to exchange information between instances of the system. As discussed in §2.4, the majority of attacks in the Internet are undirected and automated. In order to take on these types of attacks at different points of the network, communication mechanisms have to be adopted that allow quick propagation of information.

In the following, we describe three important aspects of a communication sub-system: asynchronous communication, degree of state propagation, and different communication schemes.

Asynchronous Communication

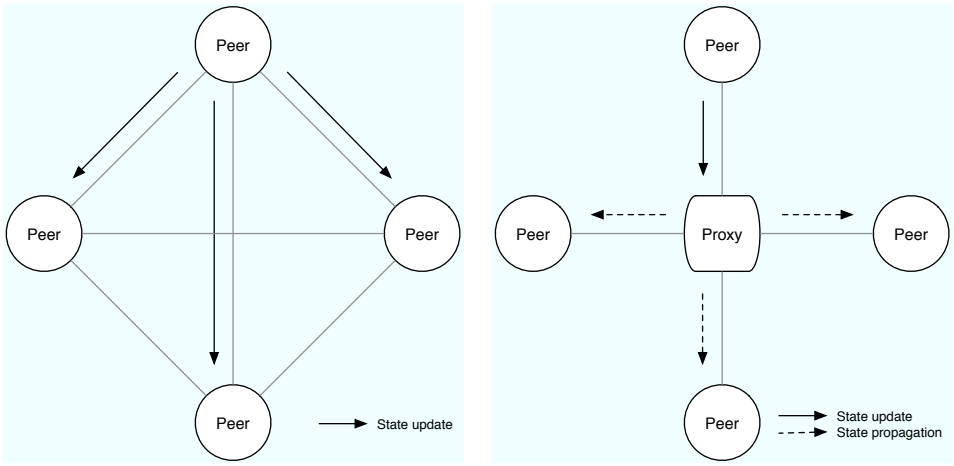
While synchronous communication is reliable, real-time constraints often inhibit such communication models [Neu94, TS02, Som05]. For example, bidirectional communication usually implies to deal with unreceived replies, requiring failure-recovery. If one instance waits for an answer but the remote side is not available anymore, it would keep the waiting instance from processing, possibly leading to packet drops. This essentially means that communication has to be *asynchronous* to meet the real-time requirements. In this unidirectional model, messages that come in at the remote side usually elicit an event which activates an event handler executing the corresponding code. If the incoming event has low priority, it can be scheduled and processed later together with another event. This type of event handling is called *batch-processing* and can ameliorate the system performance.

However, asynchronous communication entails also disadvantages. Messages can be lost in an unreliable communication channel. When a NIDS does not receive an important event notifying about an attack, the effectiveness of the system is derogated.

State Propagation

Flexible exchange of decision context among instances of a NIDS is requirement for transparency. More precisely, a much higher degree of transparency can be achieved with fine-grained control over the internal state of a NIDS.

Figure 3.1 Possible communication schemes of NIDS instances.



(a) Mesh communication. (b) Proxy communication.

Unfortunately, most of today's NIDSs lack the capability to share the internal fine-grained state which remains volatile in the memory of the NIDS. Often, only aggregated high-level state already having been coined with the site's policy can be exchanged. For example, aggregated alarm logs cannot be decomposed into their actual sources. But the granular ingredients for a triggered alarm might be an important information for other NIDS instances as well. With the condensed logs, however, other instances can only guess what actually triggered the alarm.

To overcome this limitation, the *independent state* framework (see §2.3.2) has been developed, allowing us to propagate the bulky fine-grained state between instances of a NIDS. In our work, we inherently rely on independent state to build a flexible NIDS cluster.

Communication Schemes

Different communication models are expedient depending on the expected cluster scale. If only a few instances communicate, a *meshed* scheme is the most performant. However, if the number of nodes is growing, a full mesh with $\frac{n(n-1)}{2}$ connections implies non negligible communication overhead. To reduce the overhead, each instance connects to a *proxy* broadcasting the state information to every node except the originator. Thus the communication channels are reduced to n connections. The two communication schemes are shown in Figure 3.1.

We demonstrated mechanisms for transparent load-balancing in this section. Several aspects play an important role when designing a NIDS cluster for high-speed environments. First, load-balancing offers a viable mechanism to thwart processing bottlenecks. We identified four possible traffic division schemes to distribute the work load among a

set of nodes. Thereby, we tear apart connected information in the network traffic stream and have to employ communication to make the lacking context available to the other instances of the NIDS cluster.

3.4 Bro Cluster

The following section covers our concrete realization of a distributed NIDS with the previously discussed goals and mechanisms in mind. Throughout this section we present the *Bro Cluster*, an array of machines running the Bro NIDS (see §2.3). The main reason why we selected the Bro NIDS is that it has already been in use for a long time at LBNL. We further employ the Bro NIDS because *(i)* we can greatly leverage its communication framework, *(ii)* we collaborate with people having long operational experiences with Bro and actively develop the system's core enabling us to quickly fix discovered bugs, and *(iii)* no other open-source NIDS LBNL encountered provides such a flexible configuration.

Practical issues motivated us to rethink the current operational situation. Although Bro serves its purpose well, the high traffic volume causes load problems. One single Bro cannot cope with the induced load anymore, which leads to a cleft setup: Two Bro instances run separately on different machines; one dedicated to HTTP traffic analysis only and the other is responsible for the remaining network traffic. This "crude" approach to handle the high system load clearly lacks flexibility and transparency (see §3.2.1). No intra-Bro communication to exchange valuable information can be leveraged to expand the system's decision context.

We therefore use the Bro cluster to conduct effective network intrusion detection high-performance environments, eliminating the current operational drawbacks.

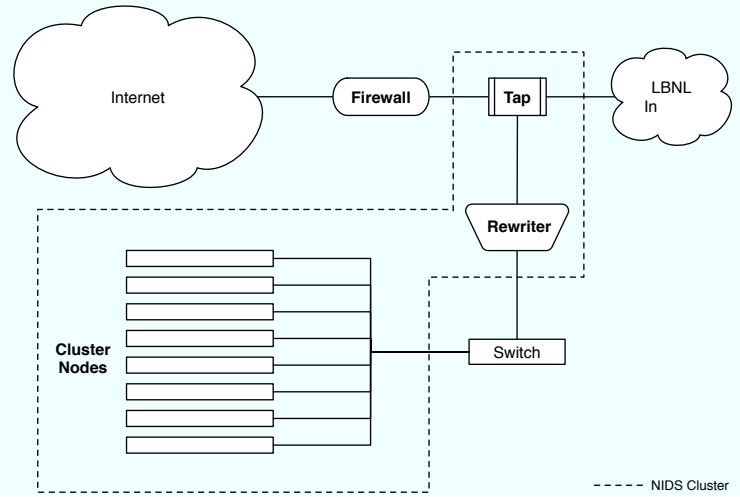
We start with a brief description of our target environment in §3.4.1, where the Bro cluster is now in operational use. In §3.4.2 we detail the hardware configuration at LBNL. We introduce our enhancements and changes to the Bro NIDS in §3.4.3 and finally summarize the benefits of the Bro cluster in §3.4.5.

3.4.1 Lawrence Berkeley National Laboratory

Our target environment is the Lawrence Berkeley National Laboratory (LBNL, [LBL]). It is the oldest of the U.S. Department of Energy's (DOE) national laboratories, managed by the University of California, Berkeley (UCB, [UCB]). LBNL has around 3,800 employees and conducts unclassified research across a wide range of scientific disciplines, mainly focusing on fundamental studies of the universe, quantitative biology, nanoscience, new energy systems, and environmental solutions. The network infrastructure comprises around 4,000 users and 13,000 hosts. The internal backbone has a capacity of 1 Gbps and DOE's Energy Sciences Network (ESNet) provides an 1 Gbps upstream link which will be soon upgraded to 10 Gbps.

LBNL uses the Bro NIDS to monitor the network traffic at various locations, particularly the external link and its DMZ. The firewall rules are in general very liberal.

Figure 3.2 LBNL load-balancing front-end.



However, a few critical hosts experience stricter rules. Leveraging Bro's flexible scripting language, LBNL incorporates a mechanism to automatically drop connectivity of attacking hosts.

The external traffic amounts to 35 TB per month, whereof 11 TB are incoming and 24 TB outgoing (36/78 Mbps on average).[2] Since the major fraction of the traffic is HTTP traffic, the HTTP protocol analysis has moved to a dedicated machine, disburdening the main processing. Both machines receive the same amount of traffic but employ different BPF filter expressions to extract the relevant portion of traffic. This poor man's load-balancing has been replaced by our deployed Bro cluster.

3.4.2 Cluster Hardware

A reliable hardware infrastructure is necessary in order to perform viable distributed network intrusion detection. Therefore, the underlying hardware architecture has to be designed with resilience in mind to meet the dynamics of large-scale networks. A flexible hardware setup permits a quick amplification of processing power if resources turn out to be insufficient. The deployed cluster hardware at LBNL incorporates these ideas and is hence suited for operational use.

In the following, we briefly describe the key elements the cluster architecture consists of, shown in Figure 3.2.

Tap. To direct a copy of each packet to the detector, an optical tap forwards a duplicate of each packet to the *rewriter*. It is an ATM fiber tap from NetOptics [Net], designed for high-performance fiber monitoring.

[2] These numbers have been recorded in 2005 [Som05]. At that time, HTTP, SSH, and FTP-DATA were the most prevalent application layer protocols.

Rewriter. The rewriter is the cluster processing *front-end* and performs hash-based traffic division by classical flows (see §3.3.1). A hash value h from the connection 4-tuple is calculated and taken modulo the number of nodes n, as denoted in 3.1.

$$h(src\,IP, dst\,IP, src\,port, dst\,port) \mod n = i \tag{3.1}$$

The rewriter is implemented as a custom kernel module. At first, each packet from the network interface card (NIC) is delivered to the kernel process. Thereafter, the hash from equation 3.1 is generated. The resulting number i maps to a particular cluster node's MAC address. Finally, the the packet's destination MAC address is rewritten and then sent to the switch. We refer to the traffic flows directed to a particular node as *slice* [KVVK02]. The custom kernel module runs on a FreeBSD machine equipped with a Pentium 4 2.60 GHz and 1 GB of memory.

Cluster nodes. We use currently 9 cluster nodes, each equipped with two Pentium III (Coppermine) 1 GHz, 3 GB of memory. On each node runs one instance of the Bro NIDS in version 1.1 with our cluster enhancements. We refer to two communicating Bro instances in the following as *peers*.

In our setup, one node will assume the role of a *proxy* node (see Figure 3.1(b)). Since the proxy does not possess the same bulky internal state as the processing nodes, the proxy manages with less memory. On the other hand, the proxy glues together logs and hence needs a larger hard disk.

Note that this architecture is principally independent of the employed NIDS. Besides division by flows, the front-end could be implemented with other traffic division schemes as discussed in §3.3.1. This is especially interesting for NIDSs that do not need such a sophisticated communication sub-system like Bro (§2.3.2).

For example, a division scheme that divides traffic by IP space will most likely not need to use inter-node communication for protocol analysis. Nevertheless, correlation over different subnets involves communication between the nodes.

3.4.3 Bro Configuration

Our objective is to run the Bro NIDS on the cluster. To this end, we have to adapt its configuration to potentiate the exchange of state information between the cluster nodes. Having the independent state framework as flexible communication sub-system in place, we need to iterate over the policy scripts and identify state that has to be exchanged. During this process, we discovered that the communication framework requires some tuning to work reliably. Fortunately, we could directly instruct the developers to enhance the communication framework with our suggestions. Our changes and contributions are now part of the official Bro distribution.

Bro consists of a layered architecture (see §2.3). Its policy-neutral core layer is inherently connection-based and hence does not result in any problems with regard to our load-balancing approach. We could principally synchronize core events, but do not need

to rely on this feature, as core events are congruent with the traffic division scheme we employ.

The policy layer consists of policy scripts that can be divided into two classes. On the one hand, there are scripts performing *intra-connection* analysis. These policy scripts operate only on data gathered from one flow. On the other hand, some scripts correlate information across multiple connections, they perform *inter-connection* analysis.

Since we use traffic division by flows, the first type of scripts should work out of the box. Yet inter-connection scripts do not have the decision context available when a dependent flow is analyzed by another peer. To mitigate this lack of context, we *synchronize* the corresponding variables by sending state information to the other peers. Thus every peer obtains the *full* decision context while processing only a subset of the entire traffic.

Core events and *user data* can be synchronized and both rely on the same serialization framework. While core events are generated at the event layer, user data are defined in policy scripts at Bro's policy layer. We abandon the possible low-level event propagation, as synchronizing user-data causes less overhead, and hence appears to be more promising for large-scale installations [Som05]. Since we further do not need to synchronize connection-based core events, we concentrate on the script layer. To synchronize user data, a script-level variable has to be declared as `&synchronized`. Any modifications to the variable will then be sent to the other peers. As a explicit example,

```
distinct_peers: table[addr] of set[addr] &synchronized
```

propagates any changes to the content of table `distinct_peers` with the effect that the variable has the same value at each peer.

In the following, we exemplary discuss the problems encountered during our examination and provide adequate solutions where possible.

Intra-Connection Analysis

Scripts performing intra-connection analysis extract their information from a single connection. The SMTP and POP3 analyzer, for example, perform analysis with a connection as unit of analysis. In general, connection-based scripts do not need any modification to work in a cluster that incorporates traffic division by flows.

But not all intra-connection-based scripts worked out of the box. For instance, we had to modify the HTTP policy script. A single TCP connection generally accommodates one HTTP connection. However, Bro has a notion of a HTTP *session* based on source and destination IP tuple rather than on a TCP flow. The idea behind indexing a HTTP session by IP tuple is to collapse all individual activities between two hosts into a single instance of surfing. This notion is clearly not TCP connection-based and may result in a HTTP session scattered across many peers. As HTTP is the most predominant application protocol, it involves a lot of state exchange in a cluster environment and produces *race conditions*[3] which reduce the accuracy of the system. We therefore

[3] We will discuss *race conditions* and their impact on the system in §3.4.4.

changed the concept of a HTTP session to a TCP flow-based notion and now achieve the same accuracy a single Bro accomplishes.

Most scripts based on intra-connection analysis hold state within a single flow but sometimes also keep track of additional information. For example, the POP3 script has a table `pop_connection_weirds` keeping track of erroneous POP3 replies, indexed by the connection originator:

```
global pop_connection_weirds:
        table[addr] of count &default=0 &read_expire = 60 mins;
```

Although the POP3 analyzer performs general protocol analysis within a single flow, this table counts errors that can occur in multiple flows. To retain the same table entries as in a single setup, we have to append the `&synchronized` attribute to the variable declaration.

Another issue we encountered on intra-connection analysis was the allocation of protocol session identifier. Such IDs usually consist of a special character with a consecutive number. A typical HTTP session ID has the following format: `%233`. To ensure unique IDs in the cluster, we could synchronize the counter variable. However, the two or more peers can simultaneously increment the synchronized variable and if the state update arrives too late at the other peer, it will use the same value. We experienced this quite often and thus decided to prefix IDs with their (hopefully) unique hostname to avoid the assignment of duplicate IDs. A positive side effect is that we can now trace back the origin of the connection after a centralized fusion of log files.

Inter-Connection Analysis

Scripts employing inter-connection analysis correlate information across multiple connections. The most vivid example is Bro's scan-detector `scan.bro`, detecting various types of scanning activities:

Backscatter. Backscatter is "unsolicited traffic that is the result of responses to attacks spoofed with a network's IP address" [PYB+04].

Address Scan. Address scans or *horizontal* scans graze multiple hosts in a specific IP range, whereas only a single service is probed.

Port Scan. A port scan or *vertical* scan tries to enumerate a high number of ports of a single machine. On the other hand, a *horizontal* scan or *address scan* probes a single service on multiple machines. The scan detector script records both kinds of activities and even distinguishes vertical scans further in *low port trolling* (privileged ports lower than 1024) and general port scans.

Password Guessing. Password guessing refers to randomly trying various username/password combinations in order to successfully login by chance. It is usually a very noisy attack and many applications already provide protections against it. However, the scan detector recognizes multiple login attempts from the same source address and raises an alert when a customizable threshold is transgressed.

Figure 3.3 Excerpt of Bro's scan detector `scan.bro`.

```
# Index by scanner address, yields the number of distinct ports scanned
global distinct_ports: table[addr] of set[port]
    &read_expire = 10 mins &expire_func=port_summary &synchronized;
# Indexed by scanner address, yields a table with scanned hosts (and ports)
global scan_triples: table[addr] of table[addr] of set[port] &synchronized;

# Coarse search for port-scanning candidates: those that have made
# connections (attempts) to possible_port_scan_thresh or more
# distinct ports.
if ( orig !in distinct_ports || service !in distinct_ports[orig] )
        {
        if ( orig !in distinct_ports )
                {
                local empty_port_set: set[port] &mergeable;
                distinct_ports[orig] = empty_port_set;
                }

        if ( service !in distinct_ports[orig] )
                add distinct_ports[orig][service];

        if ( |distinct_ports[orig]| >= possible_port_scan_thresh &&
                orig !in scan_triples )
                {
                local empty_table: table[addr] of set[port] &mergeable;
                scan_triples[orig] = empty_table;

                add possible_scan_sources[orig];
                }
        }
```

The scan-detector gathers its information from multiple connections that can possibly be analyzed by other peers. Since the detection mechanism of the scan detector inherently relies on threshold transgression checks, it is very important that the detector includes every single scan to finally raise an alert when a significant number of scans exceeds a threshold. For example, Figure 3.3 illustrates the classification of possible scan sources. It is a very coarse classification as it only considers distinct ports independent of the destination.

Consider the concrete example in Figure 3.3, where the threshold to raise an alert is 50 scanned distinct ports in 10 minutes (`possible_port_scan_thresh = 50`). If one scan probe is accommodated in one connection, the 50 connections can be evenly scattered over all analyzing peers. Without synchronization, each peer would not be able to recognize when the "global" threshold is transgressed. We declared the table `distinct_ports` as `&synchronized` to solve the problem; any change to the table by one of the peers is now propagated. Hence every peer identifies the host as a possible scan source.

3.4.4 Practical Hurdles

During our examination of the policy scripts and testing cycles, we discovered various practical hurdles. So far, we assumed that synchronizing a variable works as expected: mutually-exclusive data structures should ensure that only one peer at a time has access to a particular variable. In practice, real-time requirements impede this model, as we will describe in the following. We further identified common pitfalls throughout the conversion of policy scripts and present practical solutions where possible.

Race Conditions

Albeit the existence of synchronized data structures, various types of race conditions may emerge.

Due to real-time requirements, Bro explicitly allows the occurrence of race-conditions. The independent state framework implements synchronized tables by propagating modifications to data as operations[4]. In some circumstances, this can lead to race conditions. For example, if two peers share a synchronized table which counts alerts by source address, then each of them might detect independently that the same source address has generated an alert. When both peers now modify the table simultaneously, a race condition can occur. The winner will overwrite the loser's value, with the effect that only one value from the same source address will be taken into account. An example is given in Figure 3.4(a). Peer A and peer B generate `10` respectively `32` alerts for the same source address `x` simultaneously. By propagating the absolute value of the variable, the peers reciprocally overwrite their previous value which entails incorrect values on both sides.

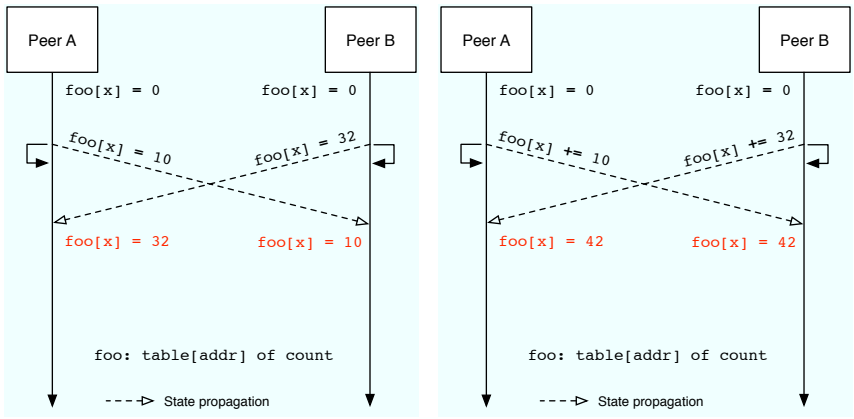
To avoid this kind of race conditions, mutually-exclusive data operations would be required. This is practically impossible as it would violate Bro's inherent real-time exigencies that forbid any type of locking. Therefore, Bro features explicitly a model of *loose synchronization* where race-conditions may occur. The occurrence of race conditions cannot be suppressed but at least mitigated. To this end, we present one special case from [Som05] and then our new additional mitigation strategy which we call *mergeable* tables.

The existing special case would be to perform "relative" operations instead of propagating absolute values whenever possible. If a state update in the above example would increment the counter of the table instead of assigning an absolute value, the counter would be increased by two, yielding the correct result. Figure 3.4(b) shows relative operations with the `+=` operator resulting in a correct common value of `42`. In this case, we did not remove the race condition but alleviated its impact.

Still, more harmful race conditions can occur. Consider the case when multiple peers modify the *inner* set of a nested table. For example, a delete operation on the inner set is automatically a modification of the corresponding index of the outer table and triggers an update for this index. The update arriving at other peers overwrites the

[4] With fully independent data, changes can be propagated in terms of descriptions of the operations to perform on the data, rather than the full bulky data itself. Examples for operations are increment/decrement counters or add/delete table entries.

Figure 3.4 Different types of race conditions.



(a) Absolute value propagation. (b) Relative operations.

Figure 3.5 Code snippet illustrating the `&mergeable` attribute.

```
global foo: table[addr] of set[port] &synchronized;
local bar: set[port] &mergeable;
add bar[y];
foo[x] = bar;
```

entire data associated to the outer index. If the peer receiving the state update has made modifications to its inner set, these modifications would be overwritten by the incoming data and are lost.

We developed a way of thwarting this race condition. To this end, we introduce the new variable attribute `&mergeable`[5]. Mergeable sets do not overwrite each other on incoming state updates, but rather build the *union* of their contents. We illustrate such a process on the basis of the previous example in Figure 3.5, but slightly modified the table `foo`. It is now a nested table with an inner set of ports. The set of ports could represent a number of distinctively scanned ports from a given source address, as shown in Figure 3.3. We then allocate a local set `bar` with the attribute `&mergeable` and add an element `y` to it. Thereafter, we bind the set `bar` to the index of the table `x`. Now consider an incoming state update `foo[x] = baz` which would overwrite the value for index `x` if we did not specify the attribute `&mergeable`. But instead, the union of the old and new set is built, i.e. the two sets are merged: `foo[x] = bar` $\cup$ `baz`.

In order to send a state update for a given variable, its value has to be referenced by a name. The values of global variables are clearly denominated by their names. However, values that are inner part of nested data structures do not possess a name. Hence Bro

[5]The Bro language features variable *attributes* which assign further properties to a variable. Attributes are specified when declaring a variable.

devises an internal name for these values, not noticeable by the user. Together with the first state update for a given value, Bro sends its corresponding name. The receiver then knows how the value is called and adopts the same name for future updates. But if two peers simultaneously generate a value for the same table entry, they both independently devise a name and send it along with the value. The mergeable attribute is responsible for consolidating two values, but what should be the future name of the result? In fact, both names represent the merged result. Technically, the incoming name is installed as an alias for the already existing name. Based on our findings, Bro has now been extended to correctly handle the fusion of tables.

Common Pitfalls

Beside race conditions, we identified other recurring problems and common pitfalls throughout the conversion of policy scripts. Having identified these pitfalls, we propose techniques to mitigate their impact.

Synchronized variables in expire functions. The capability to propagate state has to be utilized with care. Especially when using *expire functions*, unforeseen errors can arise. Expire functions are generally employed when certain actions have to be performed in conjunction with the expiration of a table element; they can contain arbitrary code. If an expire function is associated with a table, the expiring table element will be first handed to the expire function before it will be deleted.

We examine now an example given in Figure 3.6. Here, a global counter `foo_cnt` corresponds to the number of elements in the table `foo` and is decremented each time an element expires. Furthermore, the table `bar` is semantically associated to the table `foo`, because the expire function of table `foo` decrements the corresponding index of table `bar`.

Yet very subtle complications may occur in a distributed setup when peers share synchronized variables. In the example shown in Figure 3.6(a), the global variables are synchronized and any changes to them are propagated to the other peers. After the decrementation of `foo_cnt` on one side, the local change is propagated, decrementing the remote counter, too. However, the remote side expires the same element as well, resulting in another decrement operation. Altogether, the synchronized counter has a value of $n-2$ instead of $n-1$, which can lead to possible *count underflows*. On the other hand, an incoming delete operation for `bar[idx]` has no effect on a table which does not accommodate the index `idx`, unless it has been inserted again in the meantime.

Hence, to suppress the duplicate state updates, we suggested to introduce two new functions, `suspend_state_updates` and `resume_state_updates`, to suspend potentially dangerous operations in expire functions. Figure 3.6(b) depicts the use of these two functions.

Splitting multi-dimensional tables. Several tables use *multi-dimensional* indices. The scan detector, for example, contains the set `distinct_peers: set[addr, addr]`, which

Figure 3.6 Protecting a synchronized variable in an expire function.

```
global foo_cnt &synchronized;
global foo: table[addr] of port &synchronized
        &create_expire = 10 mins &expire_func=foo_exp;
global bar: table[addr] of count &synchronized
        &create_expire = 10 mins;

function foo_exp(t: table[addr] of port, idx: addr) : interval
{
--foo_cnt;
delete bar[idx];

return 0 secs;
}
```

(a) Unprotected expire function.

```
function foo_exp(t: table[addr] of port, idx: addr) : interval
{
suspend_state_updates();
--foo_cnt;
delete bar[idx];
resume_state_updates();

return 0 secs;
}
```

(b) Protected expire function.

keeps track of distinct addresses scanned by a source. To count the number of distinct destinations, the traditional method was to employ a separate table, e.g. `num_distinct_peers: table[addr] of count &default=0`, and automatically increment/decrement the address counter on every insert/delete operation. Although the second table does not expose race conditions because only increment/decrement operations are propagated, additional synchronized tables generate further state updates and should be avoided in order to minimize the total amount of exchanged state.

Thus, we suggest splitting multi-dimensional tables into nested tables. The table `distinct_peers: set[addr, addr]` is in this case converted to `distinct_peers: table[addr] of set[addr]`. First, we can now simply determine the size of the set with Bro's cardinality operator: `|distinct_peers[qux]|` yields the number of scanned hosts by `qux`, without the need for an additional table.

Second, we gain a further advantage as we can now achieve more granular expire semantics by setting additional expire timers on inner tables as well.

Threshold checks. Many activities are believed to be malicious if repeated multiple times. To count the actual occurrences of an activity, a NIDS usually employs a counter, generating an alert upon transgression. For example, a NIDS can interpret multiple login failures as password guessing. Often, threshold checks are implemented as

follows:

```
if ( ++foo_cnt == threshold )
        raise alert;
```

Synchronized data structures can, however, exhibit unexpected behavior in distributed environments. Consider the case where the counter `foo_cnt` is incremented by an incoming state update before the script reaches the code again. If `foo_cnt` had, due to the incoming state update, the same value as `threshold`, the counter would first be incremented and checked for equality. The condition would thus never be true, yielding a *false negative*.

Although the ability to modify variables remotely via state updates offers great flexibility, in this case it falls short to keep the policy scripts in tact. As previously discussed, high-speed intrusion detection has real-time requirements, prohibiting mutually exclusive data structures that can potentially defer or even block packet processing. Therefore, we have to convert existing policy scripts in a manner that they become more robust against external modifications.

3.4.5 Summary

Our motivation was to conduct efficient load-balancing in high-performance environments. We identified transparency and scalability as two major objectives of transparent load-balancing. Transparency enables us to keep a NIDS tractable, whereas scalability paves the way for growth in various dimensions. To substantiate these goals, we presented communication and load-balancing as practical mechanisms. While a flexible communication sub-system is an inherent requirement for efficient inter-node communication, load-balancing is an essential tool to distribute analysis in order to cope with the load that high-volume links induce.

We further presented the *Bro Cluster*, an array of machines performing parallel in-depth analysis of a sliced network packet stream. With independent state, we have a flexible communication sub-system at hand that we leverage to create a transparent cluster. The Bro cluster is now operationally deployed at LBNL's border and has substituted the hitherto manually coordinated Bro instances.

4 Cluster Evaluation

To substantiate the feasibility of our enhancements, we examine the performance of the cluster in various aspects. In detail, we scrutinize the resource consumption by looking at the inter-peer communication. Clearly, the most interesting question is the dimension of scale we can achieve by tossing in as many nodes as possible. Not only does the size of the cluster give interesting insights, but also more subtle aspects like the load distribution between the two Bro processes, the detection rate loss due to synchronized processing, and the overhead of the communication. Given the dynamics of live traffic, a reliable testbed to accomplish reproducible measurements is necessary. Therefore, we introduce our examination methodology in §4.1, followed by our testbed in §4.2. We audit each of the sketched evaluation facets in §4.3.

4.1 Methodology

Due to the natural dynamics of Internet live traffic, we cannot simply run our test scripts with a live packet stream to perform measurements yielding reproducible results. A common method of examining the performance of NIDS is to capture a packet stream as a `pcap` trace and feed the NIDS offline with it. Thus, the NIDS's input is invariant while the output now depends on the configuration parameter. Trace-based analysis entails a "compressed" analysis time which is referred to as *trace time* [SP05]. Yet, the communication subsystem which is independent of the trace time operates in *realtime*.

In order to perform reproducible measurements, we have to incorporate the trace-based analysis. Fortunately, we can leverage Bro's *pseudo-realtime* mode which synchronizes realtime and trace time. Bro can insert processing delays while receiving packet input from a trace. These delays coincide with inter-packet arrival gaps normally observed during packet capturing. Internally, Bro defers packet processing until the time difference to the next packet timestamp has elapsed.

In practice, the cluster processing *front-end* (see §3.4.2) divides the packet stream into flows and directs each slice to the corresponding node. Yet for our measurements, we instrument the cluster with traces to achieve reproducibility. We captured one trace and manually split it up to simulate the front-end. To slice one big trace into several small pieces, we use `tcpdump` [TCPb]. However, the filter expressions of `tcpdump` lack the modulo operation which the front-end uses to split the packet stream into flows. Thus we have to "emulate" the modulo operation. Let c be all necessary information to identify a connection, let $n \in \{m \in N \mid m \geq 2\}$ be the number of nodes, and let $r \in [n-1]$ be a possible *residue class*. Then, to obtain the residue class of a trace file, we emulate the modulo operation as follows:

$$c - \left\lfloor \frac{c}{n} \right\rfloor \cdot n \quad = \quad r \tag{4.1}$$

For example, the information c identifying a TCP connection is the 4-tuple of IP source/destination address and source/destination port. A `tcpdump` filter expression for such a connection would be:

```
ip[14:2]+ip[18:2]+tcp[0:2]+tcp[2:2]
```

4.2 Testbed

For all of our measurements, we used the same trace. We captured a one-hour packet trace in the LBNL environment. To concentrate on the relevant traffic for the analysis, we applied a BPF filter to prefilter packets we do not analyze anyway.

The volume of the trace was 11 GB and we did not experience any packet drops. We explored further details of the trace with the free trace analysis tool `tcpdstat` [Tcpa]: the trace contained 16,519,701 packets accommodated in 363,876 flows (avg. 45.40 pkts/flow) with an average rate of 28.34 Mbps. We observed a peak rate of 121.63 Mbps. 99.26% bytes of the trace were TCP traffic, only 0,74% UDP. The most prevalent protocol was HTTP (87.12%), followed by HTTPS (6.89%), SMTP (3.31%) and *icecast* (1.17%), a streaming media server supporting Vorbis and MP3 audio streams [Ice].

Throughout our measurements, we exclusively used the proxy communication scheme that was introduced in §3.3.2. Although we could have used a meshed setup for a small number of peers, we decided to retain one scheme in order to obtain comparable results.

To assess the scalability of our implementation and reduce side-effects to a minimum, we conduct several measurements with a different amount of peers. Thereby, one node always serves as a proxy, whereas the number of peers changes from 2, 3, 5, and 8.

4.3 Measurements

In this section, we discuss our measurements with regard to accuracy and performance. At first, we investigate the accuracy of the cluster in §4.3.1. Without satisfying accuracy, all further performance related measurements do not make any sense, because the cluster would else not be suited for practical use.

After the discussion of accuracy, we turn in §4.3.2 to the performance evaluation including an analysis of CPU load, communication overhead, and assessment of scalability. Finally, we summarize our observations and results in §4.3.3.

4.3.1 Accuracy

Ideally, the load-balancing NIDS cluster recognizes the same amount of intrusions as one single instance. However, practice shows that we face several temporal constraints due to state propagation that have an impact on the accuracy. Since the detection of

intrusions is the main goal of a NIDS, an alleviated accuray would render the system ineffective. Therefore, it is important to iron out any effects impairing the detection rate of the cluster. In the following, we discuss observed issues affecting the accuracy of the cluster and present mitigating solutions where necessary.

FTP-DATA connections

Initially, we had difficulties in recognizing FTP-DATA connections as such. The FTP analyzer possesses a table that keeps track of expected FTP-DATA connections, issued by the PORT command in the control connection. A single installation has the information about the incoming data connection immediately available. However, as we operate in a distributed environment, this may not be necessarily the case. Because the FTP control connection can exhibit a different TCP connection tuple and thus a different hash value from the data connection, they can be routed to different peers.

Bro keeps track of the type of every connection (e.g. FTP, HTTP, SMTP, etc.) while it is active. A connection is labelled as soon as it has been established successfully. If FTP control connection and data connection are analyzed by separate peers, we cannot identify FPT-DATA transfers reliably because of the lacking context. Thus we *synchronized* the table keeping track of expected data connections[1]. But Bro has still hardly a chance to label FTP-DATA connections successfully in time, because state propagation imposes a certain latency. Hence, we added an addional labelling when connections are recorded to disk in order to correct previously wrong labelled connections.

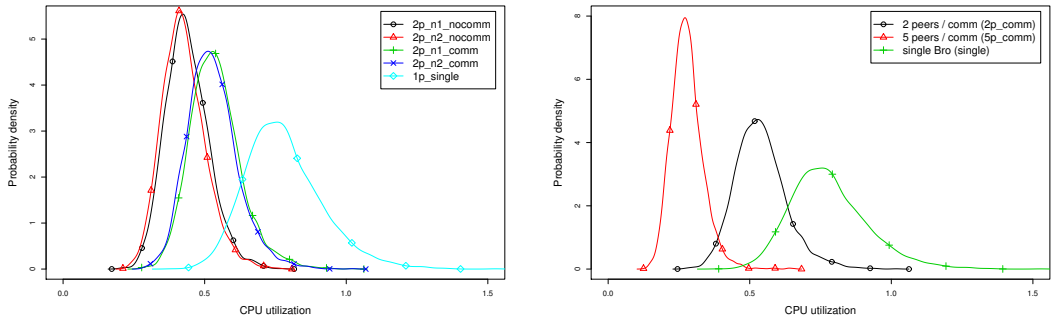
Interestingly, we still recognize only 95,32% of all FTP-DATA connections. The average duration of the missing FTP-DATA connections is 0.15 seconds. Clearly, the time window in which a state update could arrive in time is much too small. The longest FTP-DATA connection we missed lasted 0.45 seconds, meaning that we recognize FTP-DATA connections that last longer than half a second. Nevertheless, we can restore the lacking label. One solution would be to post-process the centralized logs. As the `ftp.log` contains the FTP control connection and therby the connection tuple of the expected data connection, a simple script could relabel unidentified data connections. Another solution would be to employ a different traffic division scheme (see §3.3.1) that forwards control connection and data connection to the same node.

Scan Detection

The largest fraction of inter-peer connectivity results from Bro's scan detector, containing many frequently updated tables which comprise the majority of state updates. It is therefore important to verify the accuracy of this key component. Further, converting the scan detector was the most challenging task in our work. We had to fundamentally change the data structures which the analyzer employs to keep track of scanners. We came up with a solution that is suited for concurrent processing and eleminated all awkward problems from the former single-architecture version.

[1] Technical details of our synchronized cluster configuration are explicated in §3.4.3.

Figure 4.1 CPU load of the analyzing peers (main process).



(a) 2 peers. (b) 5 peers.

Our output confirms that scan detection executes accurately in the cluster environment. The trace contains four port scans which generated alerts at 24, 43, 305, and 1734 scanned ports in the single Bro setup[2]. In the Bro cluster we achieved the same results, except for the last alert yielding only 1733 scanned ports, thereby missing one port. As these ports have been scanned rapidly in a few seconds (and in parallel with other scans), we believe that the sheer volume of spawned state updates caused the difference.

4.3.2 Performance

Given that the analysis is the most CPU intense task for a NIDS to perform, we now examine how well our load-balancing approach in terms of CPU costs turns out to be.

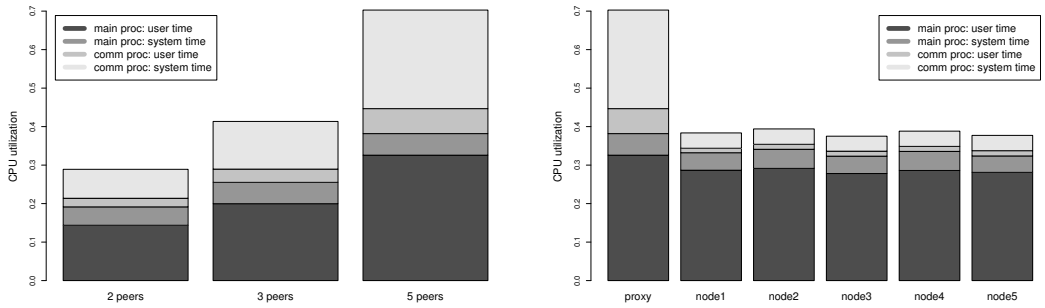
Bro starts two processes in a distributed setup. The *main process* (parent) is devoted to perform the analysis of the incoming packet stream, while the *communication process* (forked child) handles the communication with other peers. Inter-process communication is realized with a UNIX pipe: the children of different Bro instances communicate among each other, e.g. they send and receive state updates. Then, the received data are shoveled to the parent process which can integrate the state changes in its analysis, thereby broadening the decision context.

We evaluate the CPU load of both processes, but defer the examination of the communication process until we discuss the overhead of the communication sub-system. At first, we run Bro on the full trace without any communication at all. We compare all other results to this run, because a single Bro has full decision context and no result distortion due to network latency. Thereafter we use our cluster configuration from §3.4.3 containing numerous `&synchronized` variables.

Figure 4.1 shows the probability density of the CPU utilization per second in different setups. These measurements account only the user time of the main process which

[2]These values stem from Bro's *Scan Summary* yielding the total amount of scanned ports after the particular scan times out.

Figure 4.2 CPU utilization.



(a) CPU utilization of the proxy in different setups.

(b) CPU utilization of the proxy with 5 peers.

is responsible for the connection analysis. We do not account the system time when performing peer examination because we want to assess the analysis performance without I/O operations, purely concentrating on the CPU utilization. As we will later see, the system time is proportional to the user time for the peers (see Figure 4.2(b)) and would cancel out anyway in this measurement. In Figure 4.1(a), three different Bro runs are illustrated: the curve `1p_single` shows our reference run which is one single Bro instance on the entire trace without any communication. The second run, visualized by `2p_n1_nocomm` and `2p_n2_nocomm`, was conducted with two peers not communicating with each other. Since the absence of communication entails a strong degradation of the detection rate, it is inapplicable in practice. However, it helps us to assess the third run, depicted by `2p_n1_comm` and `2p_n2_comm`, where a proxy and two peers share state information.

We now look at the fairness of the traffic division scheme. The congruence of `2p_n1_comm` and `2p_n2_comm` substantiates that our load-balancing approach is indeed fair; each peer is charged with almost the same load. Moreover, enabling communication results in a slight increase in CPU utilization due to the incoming state updates from the communication process. Because of the congruence, we pick for further measurements the CPU utilization curve of one peer representative for the entire group of peers (without the proxy).

Looking at Figure 4.1(b), we see three runs. The first and second curve, `single` and `2p_comm`, are the same as shown in Figure 4.1(a) to facilitate the comparison. The third curve `5p_comm` illustrates a Bro run with 5 peers.

We see that the average CPU utilization shrinks with an increasing number of peers: the median shifts from 0.77 (`single`) over 0.53 (`2p_comm`) to 0.27 (`5p_comm`). The standard deviation was 0.14, 0.09, and 0.06 respectively. We thus deduce that our Bro cluster can effectivly balance the load over an expandable set of peers. Compared to the achieved outcome, a marginal increase in CPU load (resulting from communication with the child process) is negligible for the main process conducting the connection analysis.

Turning to the assessment of scalability, we now regard Figure 4.2. We first consider

the proxy itself as subject of analysis and therafter compare the proxy with the peers. Figure 4.2(a) shows three bars that illustrate the CPU utilization of the proxy node. The two lower segments of each bar comprise the CPU utilization of the main process, whereas the two upper bars represent the communication process. Both halves are divided in *user time* (lower half) and *system time* (upper half).

For the communication process, visualized by the upper segment of the bars, the system time makes up the largest portion of CPU utilization since inter-peer communication primarily involves I/O and not CPU. On the other hand, the main process utilizes mostly user time as it mainly integrates the state information from the child process into his own state model. The more peers a proxy has to serve, the higher the system time of the communication process becomes. This is attributed to complex serialization model which we will dicuss in the following.

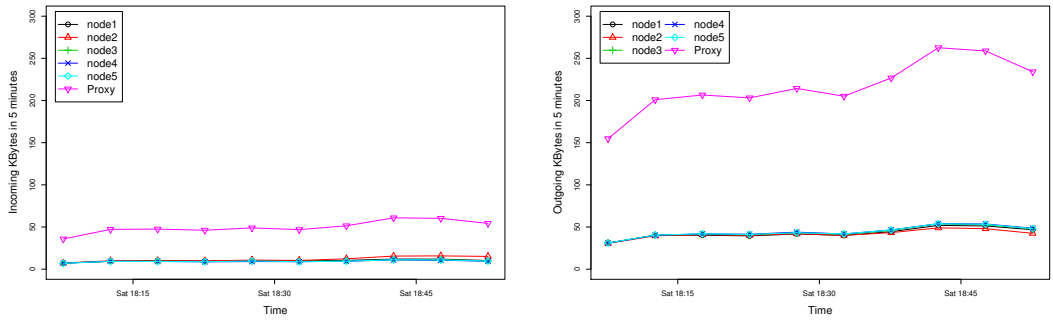
The looming question is, how many peers can we toss in until the system reaches its scalability limits? We tried to answer this question by incrementally adding peers to the setup. Surprisingly, the communication process of the proxy ran out of memory with the addition of the 6th peer. In fact, the memory capicity is sufficient on the particular nodes and the exhaustion turn out to be only a symptom of the real cause. In principle, incoming state updates are queued in a buffer until they are processed. Now if more state updates arrive than the proxy can process, the queue buffer will eventually overflow because the proxy falls short of processing.

Before dealing with the root cause of this problem, we look in Figure 4.2(b) at the unequal workload of the proxy compared to the peers. As in Figure 4.2(a), the bars split up in two segments representing main and communication process, whereas each segment again is divided in user and system time. If n represents the number of peers, we see that both, user and system time of the communication process, are n times as much as on a single peer. Note that our machines possess two CPUs and thus can run each process on a separate CPU. Nonetheless, the sheer volume broadcasted from the proxy is far too much. Looking at actual numbers, we found that the proxy receives/sends around one/five million chunks for 8 peers in 5 minutes. It is not astonishing that communication in this order of magnitude eventually causes the memory buffers to overflow.

In order to deeper understand this observation we now examine the inter-peer communication. Therefore, we present the serialization mechanism of the proxy. Considering an incoming state update of a peer, the proxy has to deserialize the update and apply it to its own in-memory state. If n represents the number of peers without the proxy, the state update is first deserialized, then serialized $n - 1$ times and sent out to the remaining $n - 1$ peers. Clearly, this way to propagate state changes is not efficient, but technical reasons forced the developers to retain this scheme. Originally the communication framework was designed to support only point-to-point connections. The current serialization framework is complex, operating at a very low level and a proxy did not fit in this model, but could be easily implemented at the expense of additional serialization costs.

Unfortunately, the scan detector generates a large amount of state updates. However, various mitigation strategies can be employed to reduce the number of state updates. Though we could simply turn off the scan detector and thereby expunging the majority of

Figure 4.3 Communication overhead with 5 peers.



(a) Incoming bytes.

(b) Outgoing bytes.

state updates, we want to conduct effective network intrusion detection at which the scan detector forms an integral part of. Another way to reduce the number of state updates would be migrating to a different traffic division scheme. For example, switching to *IP flow* division (see §3.3.1) would automatically eliminate all state updates for port scans because IP source and destination address do not change during a port scan and all corresponding packets are thus forwarded to the same peer. A different strategy would be to modify the scan detector itself so that it generates significantly less state updates. To this end, one could think of local threshold for each node: only upon transgression, inter-peer communication starts. Finally, it is possible to extend the communication scheme. By connecting multiple proxies together, a resilient hierarchical setup could be created at the cost of higher latency, but disburdening the particular proxies.

Finally, the volume of the transferred state updates is visualized in Figure 4.3. On the one hand, we consider the bytes of incoming state updates in Figure 4.3(a) and on the other hand outgoing bytes in Figure 4.3(b). Both plots have the duration of the trace on the x-axis and the transferred state updates in kilo bytes per 5 minutes on the y-axis.

Obviously the proxy does most of the work in both cases. The mean of incoming state updates from the proxy is 49.94 KB (standard deviation 7.35), whereas the mean of outgoing state updates is 216.70 KB (standard deviation 31.25). Thus, the proxy sends 4.42 times more bytes than it receives. Since every incoming state update is sent out $n - 1$ times to the remaining peers, these numbers roughly seem to be sound. In these plots, we see once again that the peers perform the same amount of work because their lines lie on top of each other.

4.3.3 Summary

Based on our observations and measurement we draw the conclusion that our NIDS cluster can effectively conduct transparant load-balancing in high-performance environments. Transparency, as discussed in the previous chapter, is achieved by the low-level state exchange of Bro. We verified in this chapter that the results provide the required

accuracy for practical appliance. Further, we investigated the performance of the cluster. Our traffic division scheme turns out to be very fair, each cluster node is charged with virtually the same load. In terms of scalability, we were able to run our cluster with 5 nodes successfully. Due to the complexity of the serialization framework, the centralized communication scheme caused the proxy to be a bottleneck. Nonetheless, we proposed various strategies to thwart this scalability limitation.

5 Conclusion

This last chapter summarizes our work. After recurring our observations and recapitulating our contributions, we sketch promising directions for future work.

5.1 Summary

The dynamics of large-scale and high-volume networks pose ambitious challenges for network intrusion detection. Traditional NIDSs reach their limits in these environments because their single-machine architecture cannot cope with the induced load. Despite closed-source solutions on expensive custom hardware, the inflexibility and limited tuneability of these systems motivate the creation of more flexible distributed systems to secure the network perimeter.

In this thesis, we present an approach to effectively conduct network intrusion detection in high-performance environments. Employing commodity hardware, we devise a cluster of communicating NIDS instances. However, the distribution of analysis entails a significant loss of decision context for each cluster node. An important challenge is thereby providing the nodes with the lacking context. Existing systems only correlate aggregated high-level information, such as logs or alerts. Contrary to these strategies, our approach originates one step lower: we leverage the flexible independent state framework of the open-source NIDS Bro to build a cluster in which each node is equipped with the same policy-neutral decision context.

To this end, we identify important concepts material to devising transparent load-balancing systems. In order to keep a NIDS tractable, *transparency* creates the impression that we interact only with a single NIDS, whereas *scalability* paves the way for growth in various dimensions. In addition, a flexible *communication* sub-system forms a key aspect for practical realization to support different types of load-balancing.

With these concepts in mind, we set out to construct a transparent load-balancing NIDS cluster ready for practical use in large-volume environments. Therefore, we introduce new functionality to the Bro NIDS which now explicitly supports cluster architectures over an expandable set of peers.

To substantiate the feasibility of our approach, we conduct accuracy and performance related measurements. By scrutinizing the resource consumption of our cluster, we gain insight about the CPU utilization and the involved communication overhead. Based on our findings, we conclude that we have an effective NIDS cluster in place that meets the challenges of large-scale networks. Our implementation is now in operational use at the infrastructure of the Lawrence Berkeley National Laboratory, protecting the network border and DMZ.

5.2 Outlook

In our work, we focused on a transparent load-balancing NIDS cluster. The next step consists in scaling a NIDS cluster beyond local site installations. Widely distributed network intrusion detection and prevention is a very important subject for future work due to an increasing global threat from automated and undirected attacks. Current Internet-scale approaches, such as DShield [DSh], turn out to be ineffectual data collection initiatives lacking the quality to produce reliable results [Som05].

In §3.2.2, we ignored geographical scalability. Yet for widely distributed network intrusion detection, geographical scale is a key requirement in order to come up with a fruitful solution. The more resources lie far apart, the higher becomes the latency. Despite asynchronous communication models, various new challenges emerge in wide-area networks. Not only does the latency increase, but also the probability of link failure mounts. If attacks are not reported promptly, network intrusion detection is significantly hindered.

Furthermore, geographical scale is strongly related to the limitations of centralized solutions which restrict further growth. Many centralized components eventually limit the system's scale due to performance and reliability issues that wide-area communication imposes. For example, a NIDS can also span organizational boundaries, demanding interprocess communication suitable for inherently unreliable wide-area networks. To increase the overall Internet security, it is indispensable to focus on a global scale. Building viable distributed systems in this order of magnitude is yet an ambitious on-going research topic.

Turning to technical aspects, elaborating on various strategies to reduce the amount of inter-peer communication is an important future goal. In §4.3.2, we propose several strategies to alleviate the communication overhead. With increasing scale, the routing scheme used to exchange state information becomes an important factor. We realized that the central aggregation of information finally reaches is scalability limits. To avoid hirarchies, sophisticated routing schemes, such as peer-to-peer schemes, qualify as a resort.

Bibliography

[ACI] ACID. `http://acidlab.sourceforge.ne`.

[Bac00] Rebecca Gurley Bace. *Intrusion Detection*. Macmillan Technical Publishing, 2000. ISBN 1-578-70185-6.

[Bis03] Matt Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003. ISBN 0-201-44099-7.

[BOG03] M. Blanc, L. Oudot, and V. Glaume. Global intrusion detection: Prelude hybrid ids. Technical report, 2003.

[Bro] Broccoli: The Bro Client Communications Library. `http://www.cl.cam.ac.uk/~cpk25/broccoli`.

[BY06] Paul Barford and Vinod Yegneswaran. An inside look at botnets. In *Proceedings of the Special Workshop on Malware Detection*, Advances in Information Security. Springer Verlag, 2006.

[CBR03] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker, Second Edition*. Addison-Wesley, 2003. ISBN 0-201-63466-X.

[DSh] Distributed Intrusion Detection System *DShield.org*. `http://www.dshield.org`.

[FGC+97] Armando Fox, Steven D. Gribble, Yatin Chawathe, Eric A. Brewer, and Paul Gauthier. Cluster-based scalable network services. In *Symposium on Operating Systems Principles*, pages 78–91, 1997.

[FGW98] Anja Feldmann, Anna C. Gilbert, and Walter Willinger. Data Networks As Cascades: Investigating the Multifractal Nature of Internet WAN Traffic. In *Proc. ACM SIGCOMM*, 1998.

[HKP01] Mark Handley, Christian Kreibich, and Vern Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proc. USENIX Security Symposium*, 2001.

[HS01] James A. Hoagland and Stuart Staniford. Viewing IDS Alerts: Lessons from SnortSnarf. In *Proc. DARPA Information Survivability Conference and Exposition*, 2001.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, 1979. ISBN 0-201-02988-X.

[Ice] icecast. `http://www.icecast.org`.

[IDM] Intrusion Detection Message Exchange Format. `http://www.ietf.org/html.charters/idwg-charter.html`.

[KMVV03] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the Detection of Anomalous System Call Arguments. In *Proc. European Symposium on Research in Computer Security*, 2003.

[KO04] Hideki Koike and Kazuhiro Ohno. SnortView: Visualization System of Snort Logs. In *Proc. ACM Workshop on Visualization and Data Mining for Computer Security*, 2004.

[Kre05] Christian Kreibich. A Graphical Environment for Analysis of Security-Relevant Network Activity. In *Proc. Usenix Technical Conference, Freenix Track*, 2005.

[KRVV04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security Symposium*, pages 255–270. San Diego, CA, August 2004.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *Proc. ACM Conference on Computer and Communications Security*, 2003.

[KVVK02] Christopher Kruegel, Fredrik Valeur, Giovanni Vigna, and Richard A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the IEEE Symposium on Research on Security and Privacy.* IEEE Press, Oakland, CA, May 2002.

[LBL] Lawrence Berkeley National Laboratory. `http://www.lbl.gov`.

[Lib] libpcap. `http://www.tcpdump.org`.

[MJ93] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Winter USENIX Conference*, 1993.

[Net] NetOptics. `http://www.netoptics.com/`.

[Neu94] Barry Clifford Neuman. *Scale in Distributed Systems*, pages 463–489. IEEE Computer Society, Los Alamitos, CA, 1994.

[Pax99] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[PN97] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *National Information Systems Security Conference*, 1997.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[PYB+04] Ruoming Pang, Vinod Yegneswaran, Paul Barford, Vern Paxson, and Larry Peterson. Characteristics of Internet Background Radiation. In *Proc. Internet Measurement Conference*, 2004.

[PZC+96] Nicholas J. Puketza, Kui Zhang, Mandy Chung, Biswanath Mukherjee, and Ronald A. Olsson. A methodology for testing intrusion detection systems. *IEEE Transactions on Software Engineering*, 22(10):719–729, 1996.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. Systems Administration Conference*, 1999.

[SF05] Lambert Schaelicke and Curt Freeland. Characterizing sources and remedies for packet loss in network intrusion detection. In *IEEE: Symposium on Workload Characterization*, 2005.

[Sky] Skype. `http://www.skype.com`.

[SMPW04] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The Top Speed of Flash Worms. In *Proc. ACM Workshop on Rapid Malcode*, 2004.

[Som05] Robin Sommer. *Viable Network Intrusion Detection in High-Performance Environments.* PhD thesis, Technical University Munich, 2005.

[SP03] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proc. IEEE Symposium on Security and Privacy*, 2003.

[SP05] Robin Sommer and Vern Paxson. Exploiting Independent State For Network Intrusion Detection. In *Proc. Computer Security Applications Conference*, 2005.

[SSMF03] Lambert Schaelicke, Thomas Slabach, Branden Moore, and Curt 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. Springer-Verlag, Berlin–Heidelberg–New York, September 2003.

[Sti03] Richard Stiennon. Intrusion Detection is Dead - Long Live Intrusion Prevention. Technical report, Gartner Inc., 2003.

[SWF05] Lambert Schaelicke, Kyle Wheeler, and Curt Freeland. SPANIDS: A Scalable Network Intrusion Detection Loadbalancer. In *Conf. Computing Frontiers*, pages 315–322, 2005.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005. ISBN 0321304543.

[Tcpa] `tcpdstat`. `http://staff.washington.edu/dittrich/talks/core02/tools/tools.html`.

[TCPb] `tcpdump`. `http://www.tcpdump.org`.

[TS02] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems – Principles and Paradigms*. Prentice Hall, 2002. ISBN 0-13-088893-1.

[UCB] University of California, Berkeley. `http://www.berkeley.edu`.

[VEK00] Giovanni Vigna, Steven T. Eckmann, and Richard A. Kemmerer. The STAT Tool Suite. In *Proc. DARPA Information Survivability Conference and Exposition*, 2000.

[VK99] Giovanni Vigna and Richard A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[VKB01] Giovanni Vigna, Richard A. Kemmerer, and Per Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In *Proc. of Recent Advances in Intrusion Detection*, number 2212 in Lecture Notes in Computer Science, 2001.

[WTSW97] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-Similarity Through High-Variability: Statistical Analysis of Ethernet LAN Traffic at the Source Level. *IEEE/ACM Transactions on Networking*, 5(1), 1997.