High-Performance IP-Accounting unter Linux

Georg Lukas
Studienarbeit

Unter Betreuung von Spiro Trikaliotis
AG Echtzeitsysteme und Kommunikation
Universität Magdeburg

18. Januar 2005

Zusammenfassung

In einem Rechenzentrum finden permanent Datentransfers statt. Um diese einzelnen Kunden zuordnen und in Rechnung stellen zu können, aber auch zur Übersicht der Infrastrukturauslastung und bei Angriffen ist ein funktionierendes Traffic-Accounting unbedingt erforderlich. In der vorliegenden Arbeit wird eine Software zum Accounting vorgestellt, die auf herkömmlicher PC-Hardware unter Linux lauffähig ist, und dabei die an einem Gigabit-Interface anfallenden Datenmengen verarbeiten kann, sowie Reserven für Angriffssituationen mit extrem hohen Paketdichten bietet.

Inhaltsverzeichnis

1 Einleitung **2**
1.1 Motivation . . . 2
1.2 Problemexposition . . . 3
1.3 Aufgabenstellung . . . 4
1.4 Ergebnisse . . . 5

2 Grundlagen und Verwandte Arbeiten **7**
2.1 Packet-Capturing . . . 7
2.2 Verbesserung des Capturing . . . 10
2.3 Weitere Performancefaktoren . . . 13
2.4 Verwandte Arbeiten . . . 14

3 Modellierung einer Accounting-Anwendung **16**
3.1 Grundlegendes Konzept . . . 16
3.2 Paket-Aufnahme . . . 17
3.3 Zwischenspeicherung . . . 18
3.4 Analyse . . . 20
3.5 Daten-Export . . . 24

4 Implementierung **25**
4.1 Schnittstelle zum Packet-Capturing . . . 25
4.2 Schnittstellen zum Datenexport . . . 29
4.3 Weitere Besonderheiten . . . 31

5 Evaluierung **34**
5.1 Tuningphase . . . 34
5.2 Produktionstest . . . 37

6 Zusammenfassung und Ausblick **43**

Listings **45**

Abbildungsverzeichnis **46**

Selbständigkeitserklärung **47**

Literatur **49**

Kapitel 1

Einleitung

1.1 Motivation

Der Internet-Boom der letzten zehn Jahre führte nicht nur dazu, dass Computer und Netzzugang heute in Firmen und in vielen Haushalten selbstverständlich sind, er erforderte auch eine stetig wachsende Zahl an Servern, um die Dienste für die Surfer, Chatter und Spieler aufrechtzuerhalten.

Die meisten dieser Server befinden sich in großen Rechenzentren mit einigen Hundert oder Tausend Rechnern. So können sie von einer gemeinsamen Klimaanlage, unterbrechungsfreier Stromversorgung und redundanter Anbindung an das restliche Internet profitieren, und im Störungsfall mit geringem Aufwand gewartet werden. Daraus ergeben sich wiederum geringe Betriebskosten, so dass es auch für Privatpersonen möglich und erschwinglich ist, einen Rechner in einem Rechenzentrum, zum Beispiel für den Betrieb eines Online-Spiels, anzumieten.

Da der Betreiber eines solchen Rechenzentrums seinen Internet-Uplink aber meist nach genutzter Bandbreite oder nach den übertragenen Datenmengen bezahlt, muss er die anfallenden Kosten für den Datenverkehr an seine Kunden weiterreichen.

Damit das möglich ist, muss an zentralen Stellen der fließende Traffic gezählt, und einzelnen IP-Adressen (und damit den Kunden) zugeordnet werden.

Die Auswertung dieser Protokolle erlaubt es neben der Rechnungsstellung an Kunden auch, Engpässe in der Infrastruktur aufzudecken und zu beseitigen, und dient so der Erhöhung der Dienstqualität. Schließlich kann man durch eine Überwachung der Statistiken auch Denial-Of-Service-Angriffe zeitnah erkennen, um gegen die Angreifer vorgehen zu können.

Es liegt nahe, die Zählung der Datenmengen den Routern aufzuerlegen,

die ohnehin alle Pakete verarbeiten müssen. Die meisten Modelle sind jedoch nicht darauf ausgelegt, statistische Daten zu hunderttausenden von Verbindungen zu speichern, und sind durch die zusätzliche Aufgabe so ausgelastet, dass sie mit ihrer normalen Arbeit nicht mehr fertig werden.

Die Anschaffung teuerer Routersysteme, die auf die zusätzliche Belastung ausgelegt sind, ist dagegen meist aus finanzieller Sicht nicht sinnvoll - und selbst da geht man das Risiko ein, dass bei einer Überlastung des Accounting die Routertätigkeit in Mitleidenschaft gerät.

Ein anderer Lösungsansatz besteht darin, an strategischen Punkten in der Netzinfrastruktur eine Kopie des gesamten dort fließenden Datenverkehrs an dafür bereitgestellte Rechner zu schicken, die dann die Auswertung übernehmen. Die Auswahl der Messstellen sollte so erfolgen, dass keine Verbindungen doppelt gezählt werden, oder diese Doppelberechnung muss beim Zusammenfügen der Daten korrigiert werden.

Wesentlich wichtiger ist jedoch die Dimensionierung der Rechner, die die Auswertung der Daten durchführen sollen. Der Datenverkehr, der über eine Gigabit-Schnittstelle empfangen werden kann, reizt einen gewöhnlichen PCI-Bus schon aus, und die Auswertung der Daten stellt auch schnelle CPUs auf eine harte Probe. Im Folgenden geht es um das Design einer Anwendung, die dieser Herausforderung gewachsen ist.

1.2 Problemexposition

Ein Rechenzentrum mit einigen Tausend Servern muss hierarchisch gegliedert werden, um die Übersicht und die Ausfallsicherheit zu steigern. Das Netz muss zum einen physikalisch durch die Verschaltung von Routern und Switchen und zum anderen logisch (z.B. nach Produkttyp der aufgestellten Server oder nach Zugehörigkeit der Rechner zum administrativen Netz) aufgeteilt sein.

Im vorliegenden Fall sind auf unterster Ebene jeweils 22 Rechner über einen managed Switch zusammengeschaltet. Die Uplink-Ports von bis zu 24 dieser Geräte sind an einen VLAN-fähigen Layer-3-Switch von Extreme Networks angeschlossen.

Dieser verfügt über zwei Uplink-Ports, von denen einer zum zentralen Core-Router geht und der andere als Mirror-Port konfiguriert ist, also sämtlichen durchlaufenden Datenverkehr ausgibt.

An diesem zweiten Port ist ein PC-System angeschlossen, das den gesamten Datenverkehr empfangen und auswerten soll. Dieses PC-System hat die Aufgabe, Statistiken über die gemessenen Datenmengen zu erstellen und aufgetrennt nach IP-Adressen an einen zentralen Accounting-Rechner wei-

terzuleiten.

Die bisher verwendete Analysesoftware IPAudit stößt bei dem anfallenden Datenverkehr (bis zu 300 Mbit/s bei durchschnittlichen Paketgrößen von 300-400 Bytes) an ihre Grenzen. Insbesondere bei Denial-Of-Service-Angriffen laufen die internen Puffer voll, und die Auswertung der Messdaten bleibt aus.

Durch die immer größer werdenden Freivolumen und die sinkenden Gigabyte-Preise ist dabei ein Weitersteigen der auszuwertenden Datenmengen langfristig absehbar.

Da Optimierungen an dem IPAudit-Code nur geringfügige Verbesserungen gebracht haben, und eine weitere Effizienzssteigerung angesichts der Programmstruktur nur schwer realisierbar erschien, sollte eine speziell auf diese Aufgabe ausgerichtete Anwendung neu entwickelt werden.

1.3 Aufgabenstellung

Im Rahmen des Betriebspraktikums war eine Software zu entwickeln, die mit möglichst geringen Paketverlusten den von einem Switch-Mirror-Port bereitgestellten und über eine Netzwerkkarte eingelesenen Datenverkehr analysiert, und darüber in Intervallen von einigen Minuten Verkehrsstatistiken erstellt.

Diese Anwendung sollte auf aktuellen High-End-PC-Systemen laufen, und auf SMP-Systemen möglichst gut mit der Anzahl der Prozessoren skalieren. Weiterhin sollte sie gegen Netzüberlastungen, z.B. durch Denial-Of-Service-Angriffe, möglichst immun sein, also ihre Aufgabe auch in extremen Lastsituationen erfüllen.

Aus diesen Anforderungen ließen sich Teilprobleme ableiten, für die Lösungen gefunden und zu einem Ganzen zusammengefügt werden mußten. Diese Teilprobleme waren:

- **Paketempfang**
  Es war ein Weg zu finden, um die Datenpakete möglichst effizient von der Netzwerkkarte, die sie von der externen Infrastruktur empfing, in die Anwendung zu befördern. Bei mehreren zehn bis hundert tausend Paketen pro Sekunde versagen dabei die Bordmittel von Linux, und man muss auf spezielle Schnittstellen zurückgreifen.

- **Analyse**
  Die empfangenen und zwischengepufferten Pakete galt es zu analysieren, und die relevanten Daten wie das verwendete Protokoll, die IP-Adresse des Kunden und die Paketgröße zu extrahieren. Dabei sollten

Pakete mit VLAN-Header, wie sie zwischen den Switchen im Rechenzentrum übertragen werden, korrekt verarbeitet werden, und das für den zukünftigen Einsatz vorgesehene IPv6-Protokoll unterstützt und korrekt ausgewertet werden.

- **Zwischenspeicherung**
  Die so pro Paket gewonnenen Datenschnipsel mussten nun in eine Struktur eingeordnet werden, in der sie beim Auftreten von weiteren Paketen mit den selben Eckdaten schnell wiedergefunden werden würden, um dort die Zähler zu erhöhen.

- **Ausgabe**
  In regelmäßigen Zeitabständen (im Normalfall immer nach einigen Minuten) sollten schließlich die Zählerstände an eine externe Anwendung ausgegeben und die internen Zähler zurückgesetzt werden.

- **Multiprocessing**
  Im Mittelpunkt der Entwicklung stand dabei die Aufteilung der einzelnen Aufgaben derart, dass sie möglichst unabhängig und skalierbar auf mehreren Prozessoren eines SMP-Rechners ausgeführt werden können.

- **Stabilität**
  Das Programm sollte auf einem vollständig ferngewarteten System über lange Zeiträume stabil laufen und regelmäßige Reports liefern. Neben besonders sorgfältiger Programmierung und der Vermeidung von Speicherlecks musste es auch eine hohe Resistenz gegen die potentiell böswilligen Eingangsdaten von Nutzerkontrollierten Rechnersystemen bieten.

1.4 Ergebnisse

Mit der im weiteren Verlauf der Arbeit vorgestellten Anwendung „flipacc" in Kombination mit dem PF_RING-Kernelpatch ist es möglich, die im normalen RZ-Betrieb an einer Gigabit-Leitung anfallenden Datenmengen auf einem x86-Server-System vollständig zu protokollieren und den sie verursachenden Kunden zuzuordnen. Damit lassen sich Auswerteraten von über 500.000 Paketen pro Sekunde erreichen, was auch die Analyse von Denial-Of-Service-Angriffen erlaubt.

Die nach den oben vorgestellten Kriterien entwickelte Software reizt dabei die Plattform und den Kernel bis zu ihren Grenzen aus und erlaubt es so, statt teuerer Spezialhardware oder Switchen mit eingebauter Zählfunktion vergleichsweise preiswerte Linux-Server zum IP-Accounting zu verwenden.

Im folgenden Kapitel werden verwandte Werke sowie Arbeiten, auf denen die Implementierung der Anwendung fußt, vorgestellt. Es wird die Effizienz unterschiedlicher Verfahren zum Empfang von Netzwerkpaketen beleuchtet, und auf einige grundlegende Performance-Engpässe eingegangen.

In Kapitel 3 geht es dann um das Design einer Anwendung zum effizienten Aufnehmen und Auswerten von Paketen. Es wird erklärt, welchen Weg die Daten nehmen, welche Algorithmen zur Zwischenspeicherung und Analyse verwendet werden, und wie die Aufgaben auf Threads aufgeteilt werden.

Mit konkreten Implementierungsdetails, also den Schnittstellen für aufrüstbare Module und dem Aufbau der Datenstrukturen zur internen Kommunikation und Synchronisation des Programms befasst sich schließlich das vierte Kapitel.

Zuletzt wird das entwickelte Programm dann im Einsatz getestet und mit anderen Lösungen verglichen. Nach einer Beschreibung des dazu verwendeten Versuchsaufbaus und einer Auflistung der Messergebnisse kommt eine Zusammenfassung und ein Blick auf die weiteren Ausbaumöglichkeiten und Zukunftsperspektiven.

Kapitel 2

Grundlagen und Verwandte Arbeiten

In einer Anwendung, die Daten von einer ausgelasteten Gigabit-Schnittstelle auswertet, ist die Leistung jeder Komponente entscheidend. In diesem Kapitel werden Werke vorgestellt, die bei der Lösung der Teilprobleme der Programmentwicklung hilfreich waren, aber auch nicht verwendete Alternativen aufgezeigt.

Den Schwerpunkt bilden Verfahren, mit denen man unter Linux alle an einer Netzwerkkarte ankommenden Datenpakete (oder auch eine auf bestimmte Weise eingegrenzte Untermenge) empfangen und z.T. vorsortieren kann. Es wird aber auch auf generelle Techniken zur Verbesserung der Performance und auf die Details von Hashtabellen als Organisationsform für Daten eingegangen.

2.1 Packet-Capturing

Um den Inhalt von Datenpaketen auszuwerten. gibt es zahlreiche Ansätze. Allen ist gemein, dass Pakete, die von der Netzwerkkarte empfangen wurden, nicht auf normalem Wege durch die Protokollschichten nach oben gereicht werden, sondern mehr oder weniger direkt an die auswertende Anwendung übergeben werden.

Es gilt jedoch zu unterscheiden, ob die Pakete an Software im Userland übergeben werden, oder ob deren Auswertung im Kernel stattfindet. Während Ersteres neben flexiblerer Methoden und besserer Weitergabe der Ergebnisse auch höhere Systemstabilität im Fehlerfall erlaubt, hat die kernelseitige Auswertung einen Performance-Vorteil – die Datenpakete müssen nicht aus dem Kernel-Speicherbereich in den Userspace übertragen werden, und es las-

sen sich Kontextwechsel einsparen.

Im Folgenden sollen die wichtigsten Capturing-Schnittstellen – zum Teil auch mit den entsprechenden Anwendungen – kurz erläutert werden. Dabei wird im Detail auf ihre Besonderheiten, Vor- und Nachteile in Bezug auf hohes Paketaufkommen eingegangen.

2.1.1 PCAP

Die auf Unix-Systemen am meisten verbreitete Schnittstelle zum direkten Empfang von Netzwerkpaketen ist die libpcap[1] (Packet Capturing Library). Ihre Verbreitung verdankt die Bibliothek der einfachen und einsteigerfreundlichen Programmierung sowie ihrer Verfügbarkeit auf vielen Plattformen.

Sie unterstützt nicht nur die Verwendung von Netzwerkkarten als Datenquelle, sondern auch von Dateien im PCAP-eigenen Format, die mit der Bibliothek natürlich auch erstellt werden können und Verwendung in zahlreichen Programmen wie Ethereal und tcpdump finden.

Nach einer Initialisierung mit wenigen Befehlen kann man entweder die Kontrolle an eine Bibliotheksfunktion übergeben, die bei jedem reinkommenden Paket eine Callback-Funktion aufruft, oder blockierend auf neue Pakete von der Netzwerkkarte warten. In beiden Fällen wird in der Linuximplementierung auf einem raw socket der `recvfrom()`-Befehl zum Datenempfang vom Kernel verwendet, der verhältnismäßig teuere Kontextwechsel und Speichertransfers impliziert.

Um den letzten Punkt auszugleichen, existiert eine inoffizielle Erweiterung – `libpcap-mmap`, bei der Pakete effizienter über einen gemeinsamen Speicherbereich übergeben werden.

2.1.2 netfilter/iptables

Der Linux-Kernel bringt bereits eine Möglichkeit mit sich, den Datenverkehr, den einzelne Rechner (bzw. deren IP-Adressen) verursachen, mitzuzählen. Die Zählung erfolgt mit Hilfe der Kernel-Netfilter-Module, die Konfiguration mit der `iptables`-Software[2].

Mit IPAC-NG[3] existiert auch eine Anwendung, die die internen Kernel-Zähler auslesen und zu Diagrammen verarbeiten oder in eine Datenbank einspeisen kann.

Die Zähler sind jeweils an eine `iptables`-Regel gebunden und erfassen die Anzahl der Pakete und Bytes, auf die diese Regel zutrifft. Die Überprüfung der Regeln erfolgt für jedes Paket sequentiell so lange bis eine übereinstimmende Regel mit einem „Target” - also einer Verfahrensanweisung gefunden

wurde. Im Durchschnitt ist also mit jedem Paket ein (kernelseitiger) Aufwand von $O(M)$ verbunden, wobei M die Anzahl der Regeln ist.

Um aber die Einzelaufzeichnung für eine vier- oder fünfstellige Anzahl an Rechnern durchzuführen, ist ein solcher Aufwand nicht akzeptabel.

Durch Zusammenfassung der IP-Adressen zu Netzen und eine baumartige Verschachtelung der Regeln lässt sich die Menge der Operationen auf $O(logM)$ reduzieren, was aber zum einen nur eine geringe Linderung des Problems ist und zum anderen ein sehr komplexes und von Hand nicht wartbares Regelwerk hinterläßt. Da das Zählen von Paketen nicht die Hauptaufgabe von netfilter ist, ist eine Implementierung einer Regelerstellungslösung zum IP-Accounting aber nicht zweckmäßig.

2.1.3 netfilter/ULOG

Neben der Möglichkeit, direkt im Kernel regelbasiert Pakete zu zählen, können diese auch vom netfilter-Code an ein Programm im Userspace übergeben werden. Diese Schnittstelle ist allerdings in erster Linie auf Firewall-Anwendungen zugeschnitten, und es gelten die selben Einschränkungen und Probleme wie beim Sammeln der Daten mit der libpcap.

2.1.4 NetFlow

NetFlow ist ein ursprünglich von Cisco entwickeltes Verfahren, bei dem ein Router den anfallenden Datenverkehr nach einzelnen Verbindungen (Flows) mitzählt. Ein Flow ist dabei eine logische Verbindung zwischen zwei Netzteilnehmern, die auf dem Router durch fortwährend gleiches Protokoll, gleiche Adressen und Ports eingeordnet wird.

Erkennt der Router, dass zu einem Flow keine Datenpakete mehr ausgetauscht werden, betrachtet er die Verbindung als geschlossen und verschickt die darüber angesammelten Informationen in einem konfigurierbaren UDP-Paket an einen Collector – einen Rechner, der die Auswertung der NetFlows (auch von mehreren Routern) übernimmt.

Dieses Modell hat den Vorteil, dass man relativ einfach Traffic-Informationen von allen relevanten Stellen in einem Netzwerk bekommen und auswerten kann. Allerdings erfordert das Mitzählen der Flows auf dem Router sehr viel Speicher und CPU-Zeit, was insbesondere in großen Netzen seine Routing-Performance beeinträchtigen kann. Außerdem ist ein ausfallsicheres Accounting damit nur schwer realisierbar.

Mit nProbe existiert auch eine Implementierung des Zählers für Linux/Unix, die zwar unter der GPL steht, jedoch nur kostenpflichtig verfügbar ist.

2.2 Verbesserung des Capturing

Die Auswertung von Netzwerkpaketen hat zwei limitierende Faktoren - den maximalen Daten- und den Paketdurchsatz. Während mit größeren Datenpaketen auch die intern zu übertragenden Datenmengen ansteigen, bringt jedes Paket, egal wie klein es ist, einen minimalen Bearbeitungsaufwand mit sich.

Für die Messung des Datenverkehrs sind aber auch bei großen Paketen nur die Header interessant. Eine gängige Optimierung besteht daher darin, empfangene Pakete möglichst frühzeitig – also unmittelbar nach dem Empfang von der Netzwerkkarte – auf 64 bis 80 Bytes zu stutzen.

Kleine Pakete sind dagegen problematischer – sie erzeugen fast genau so viel Aufwand, belegen das Medium aber nur einen Bruchteil der Zeit. Dadurch können weitaus mehr kleine Pakete in der selben Zeit verschickt werden, so dass die effektive Last sowohl beim Routing als auch beim Accounting ansteigt.

Da aber dieser Effekt gern bei Denial-Of-Service-Angriffen ausgenutzt wird, werden bei den meisten Angriffen die Systeme mit sehr vielen kleinen Paketen bis zum Zusammenbruch gebracht. Gerade im Angriffsfall ist es aber höchst wichtig, die Quelle bzw. das Ziel des Angriffs zu ermitteln, so dass die Traffic-Auswertung nicht versagen darf.

Im Folgenden werden Ansätze vorgestellt, die dazu beitragen sollen, die Systemlast bei hohem Paketaufkommen gering zu halten, und CPU-Reserven zum Auswerten nutzen zu können.

2.2.1 Device Polling

Im Normalbetrieb wird für jedes an die Netzwerkkarte addressierte Paket ein Hardware-Interrupt ausgelöst[1]. Der Kernel führt daraufhin die zuständige Treiberfunktion aus, die das Paket von der Karte empfängt, und erlaubt danach weitere Interrupts von der Karte.

Dieses Verhalten ist bei moderater Last (wenige, große Datenpakete) angemessen, versagt jedoch, wenn die Intervalle zwischen Interrupts kleiner sind, als die zu ihrer Bearbeitung benötigte Zeit. In einem solchen Fall wird sich die CPU nur noch mit der Interruptbearbeitung beschäftigen, und für die Auswertung der empfangenen Pakete bleibt keine Möglichkeit mehr.

Als Lösung für dieses Problem wurde von einigen Karten-Herstellern ein Verfahren namens *Interrupt Mitigation* eingeführt. Dabei wartet die Netzwerkkarte nach einem Paketempfang eine bestimmte Zeit auf das Eintreffen

[1] Befindet sich die Karte im Promiscuous Mode, so führt jedes empfangene Paket zu einem Interrupt.

weiterer Pakete. Wird der Timeout überschritten, wird für alle bisher angesammelten Pakete ein gemeinsamer IRQ ausgelöst. Dieses Verfahren ist jedoch sowohl kostspielig zu implementieren, da es von der Hardware ausgeführt werden muss, als auch verschlechtert es die Latenz beim Empfang. Außerdem ist die Anzahl der Pakete, die so zusammengefasst werden, durch die Größe der karteneigenen Puffer und durch die Anzahl der vorhandenen Kontrollstrukturen begrenzt.

In der 2.6.er Serie des Linux-Kernels wurde als Lösung daher die *New API* (NAPI) für Netzwerktreiber eingeführt. Der dahinterstehende Grundgedanke ist, dass sobald ein Interrupt von der Karte empfangen wurde, die Generierung von weiteren IRQs in der Karte abgestellt wird, und eine Kernel-Task damit beauftragt wird, Pakete von der Karte zu lesen. Wenn die Task fertig ist, und keine weiteren Pakete in der Karte zwischengespeichert sind, wird die IRQ-Generierung wieder aktiviert.

Ist das System im Leerlauf, ändert sich nichts gegenüber normaler IRQ-Behandlung. In einer Volllastsituation dagegen wird der Empfang von weiteren Paketen vom Scheduler geregelt, so dass auch deren Auswertung CPU-Zeit erhält. Dass dabei die Kartenpuffer volllaufen können und Pakete verworfen werden, ist dabei sogar von Vorteil, da das System für ihre Behandlung ohnehin keine Zeit hat.

2.2.2 Linux Socket Filter

Oft ist es nicht nötig, alle Pakete, die man empfängt, auszuwerten – z.B. wenn man nur an Daten von einem bestimmten Protokoll oder zu einem bestimmten Host interessiert ist. Damit nicht alle Pakete, die die Karte empfängt, den verhältnismäßig langen Weg nach oben in die Anwendung gehen müssen, hat man im Kernel die Möglichkeit eingebaut, Pakete frühzeitig durch einen Filter zu schicken, der über ihr weiteres Vorankommen entscheidet.

Das vom Berkeley Packet Filter abgeleitete Verfahren implementiert eine Registermaschine, die mit Code-Fragmenten ausgestattet eine Analyse von Datenpaketen durchführt und so entscheidet ob ein Paket weiter verarbeitet wird.

Der Code für die Registermaschine wird dabei aus textuellen Anweisungen wie „`ip6 or net 10.0.0.0/8`" kompiliert und im Kernel an einen Socket gebunden. Pakete, die daraufhin an diesem Socket ankommen, werden durch die Registermaschine analysiert und in Abhängigkeit von ihrer Ausgabe weiterverarbeitet oder verworfen.

Obwohl diese Funktion nicht auf bestimmte Socket-Typen beschränkt ist (lediglich TCP-Verbindungen werden nicht als Basis unterstützt), dient sie in erster Linie für raw sockets, wie sie von der libpcap verwendet werden.

2.2.3 Packet Sampling

Ein weiteres Verfahren, bei dem die Anzahl der ausgewerteten Pakete verringert wird, ist das Sampling. Dabei wird von N ankommenden Paketen genau eins zur weiteren Verarbeitung gegeben und die anderen verworfen. Dadurch wird die Last zwar um den Sampling-Faktor N gesenkt, es treten aber Ungenauigkeiten bei der Messung auf – die finalen Ergebnisse werden dadurch interpoliert, dass die tatsächlichen Messwerte wieder mit dem Sampling-Faktor multipliziert werden.

Auf einem stark frequentierten Netzwerk verteilen sich die Diskrepanzen zwar relativ gleichmäßig, was die Verwendung solcher Daten für die Administration ermöglicht, jedoch sind interpolierte Messwerte nicht als Nachweis gegenüber dem Kunden geeignet.

2.2.4 PF_RING

Eine interessante Arbeit zum möglichst effizienten passiven Packet-Capturing ist der `PF_RING`-Treiber[4] von Luca Deri. Ausgehend von der Grundannahme, dass die Verwendung einer Netzwerkkarte exklusiv für die Datenanalyse akzeptabel ist, wird ein Modell vorgestellt, bei dem alle auf dieser Karte ankommenden Pakete sofort in einen Ringpuffer geschrieben werden, der als shared memory einer Anwendung zur Auswertung zur Verfügung steht. Eine weitere kernelseitige Verwendung der Pakete wird unterlassen, damit die Rechenzeit für die auswertende Applikation frei bleibt.

Dadurch, dass alle Umwege vermieden werden, lassen sich deutlich mehr Pakete aufnehmen, und es bleibt mehr Zeit zum Abarbeiten dieser. Nur mit solchen Mitteln ist es übehaupt möglich, den gesamten auf einer Gigabit-Karte auflaufenden Datenverkehr in einer User-Space-Anwendung zu analysieren.

Der Ringpuffer bietet neben dem direkten Weg in den Adressraum der Anwendung auch weitere Vorteile – läuft er voll, weil die Software überfordert ist, werden die Pakete im Kernel verworfen und es bleibt mehr Zeit für die Abarbeitung bestehender Pakete. Außerdem erlaubt er eine sehr einfache Schreib-/Lese-Synchronisation.

Realisiert wird die Erweiterung mit einem eigenen Socket-Typ im Linux-Kernel, deshalb ist es auch möglich, einen Socket Filter einzubinden, der für eine Vorsortierung sorgt. Auch damit bleibt aber der Aufwand, ein Paket vom Medium zum auswertenden Programm zu kriegen, sehr gering.

Es existiert auch ein Patch für die libpcap, so dass existierende Anwendungen, die auf der Bibliothek aufbauen, von den Vorteilen des neuen Sockets profitieren können.

2.3 Weitere Performancefaktoren

Neben den unmittelbar für die Paketaufnahme relevanten Vorgängen können sich auch andere Systemeigenschaften positiv oder negativ auf die Geschwindigkeit einer Accounting-Anwendung auswirken.

2.3.1 PIC / IO-APIC

Auf den meisten Einprozessorsystemen sorgen zwei PIC 8259 (Programmable Interrupt Controller) für die Priorisierung der Hardware-IRQs. An diese muss auch das Ende der Abarbeitung gemeldet werden, was durch einen Port-I/O-Befehl geschieht. Durch ihre – Jahrzente zurückreichende – Abwärtskompatibilität sind diese Controller aber sehr schlecht angebunden, so dass bei sechsstelligen IRQ-Mengen pro Sekunde ein merkbarer Teil der Rechenzeit allein für die Portzugriffe aufgewendet wird.

SMP-Systeme und Rechner mit eingebautem IO-APIC haben dieses Problem nicht, da sorgt ein neuer Advanced Programmable Interrupt Controller für die Zuteilung von IRQs zu den Prozessoren. Da dieser über PCI-Speicherzugriffe angesteuert wird, läuft die Abhandlung von IRQs wesentlich flüssiger vonstatten.

Unter Linux lässt sich die Art des verwendeten Interrupt Controllers zur Laufzeit aus `/proc/interrupts` ermitteln. Lauten alle Einträge in der vorletzten Spalte `XT-PIC`, so ist der alte PIC 8259 am Werk. Ist bei den Performance-kritischen Einträgen, also insbesondere den Netzwerkkarten, `IO-APIC-level` oder `IO-APIC-edge` eingetragen, so ist der IO-APIC aktiv. Auf manchen Platinen für eine CPU lässt sich der IO-APIC zusammen mit dem CPU-eigenen local APIC über den Kernel-Bootparameter „`lapic`" beim Systemstart aktivieren.

Der PIC 8259 bietet zwar einen Auto-EOI-Modus, in dem die IRQ-Bestätigung mit langwierigen Portzugriffen entfällt. Dieser wird unter Linux aber nur bei gefundenem IO-APIC verwendet, so dass normale Systeme davon nicht profitieren.

2.3.2 NPTL

Die unter Linux am meisten verbreitete Implementierung des POSIX-Thread-Standards[5] ist LinuxThreads. Dadurch, dass dort jeder Thread auf einen Kernel-Prozess abgebildet wird, ist zum einen keine vollständige POSIX-Kompatibilität möglich, und zum anderen müssen Interaktionen zwischen Threads über Signale abgebildet werden.

Mit der 2.6.er Kernel-Serie wurden zwei Erweiterungen eingeführt, die bei diesen Problemen Abhilfe schaffen. Zum einen sind das Futexe (fast mutexes), die die Implementierung von schnellen POSIX-Mutexen und -Semaphoren unterstützen, zum anderen eine Erweiterung des `clone()`-Systemcalls zur Erzeugung von echten Threads.

Diese beiden Erweiterungen werden von der NPTL, der New Posix Thread Library, übernommen und an Anwendungen weitergereicht. Dabei wird beim Laden des Programms überprüft, ob ein kompatibler Kernel vorhanden ist, und die Erweiterung automatisch genutzt. So profitieren Anwendungen von der schnelleren Thread-Erzeugung und insbesondere von dem verbesserten Scheduling kritischer Bereiche, was bei vielen voneinander abhängigen Threads deutliche Verbesserungen bringen soll.

2.4 Verwandte Arbeiten

Es existieren bereits viele Programme, deren Aufgabe darin besteht, Datenverkehr zu analysieren und Statistiken in beliebiger Form auszugeben. Die meisten sind aber nicht mit dem Ziel entwickelt worden, den Datenverkehr eines großen Rechenzentrums an einem Gigabit-Interface aufzunehmen, so dass die verwendeten Algorithmen nicht mit den anfallenden Paketmengen skalieren. Außerdem wird in den meisten Fällen die libpcap als zentrale Grundlage eingesetzt, und viele der Programme werden nicht aktiv gepflegt.

Hier sollen exemplarisch die besseren Anwendungen vorgestellt werden.

2.4.1 IPAudit

Die Anwendung IPAudit bezieht Datenpakete von der libpcap oder aus einer Datei und sortiert sie nach Flows (N-Tupeln aus je zwei IP-Adressen, dem Protokoll und den Ports) in eine große Hash-Table ein. Empfängt der laufende Prozess ein Signal, werden die Daten ausgegeben.

Wenn man die Speicherung der Portnummern deaktiviert, lässt sich auf einem System mit einem Athlon XP 2400+ und einer Glasfaser-Gigabit-Ethernet-Karte normaler Datenverkehr mit bis zu 300 MBit/s protokollieren. Treten dagegen Angriffe mit vielen kleinen Paketen auf, kommt die Anwendung nicht hinterher. Dabei verbraucht sie deutliche Mengen an Speicher und schafft es nicht einmal mehr, die bis dahin angesammelten Informationen auszugeben.

2.4.2 nTop

Auch auf der PCAP-Bibliothek basiert nTop, ein Netzwerkanalysetool, das mit der Absicht entwickelt wurde, auf möglichst vielen Unix-Plattformen und auf Win32 lauffähig zu sein. Neben einem eingebauten Webinterface besitzt es die Möglichkeit, zahlreiche Netzwerkprotokolle zu erkennen und zu analysieren, sowie darauf aufbauend Statistiken anzuzeigen und zu exportieren.

Kapitel 3

Modellierung einer Accounting-Anwendung

In diesem Kapitel wird der Aufbau der Anwendung vorgestellt, und die für die Implementierung verwendeten Algorithmen beschrieben. Es wird darauf eingegangen, welche Alternativen es gibt und begründet, warum diese nicht eingesetzt wurden.

3.1 Grundlegendes Konzept

Die Daten durchlaufen in einer IP-Accounting-Anwendung drei grundlegende Phasen – zunächst werden die einzelnen Pakete von der Netzwerkkarte eingelesen und zwischengespeichert. Danach findet eine Auswertung der Pakete statt, wo die Datenmengen den entsprechenden Rechnern oder Kunden zugewiesen werden. Schließlich werden die ausgewerteten Daten im Zyklus von wenigen Minuten in einer bestimmten Form exportiert, um dann für die Netzüberwachung und – in zusammengefasster Form – Rechnungsstellung verwendet zu werden.

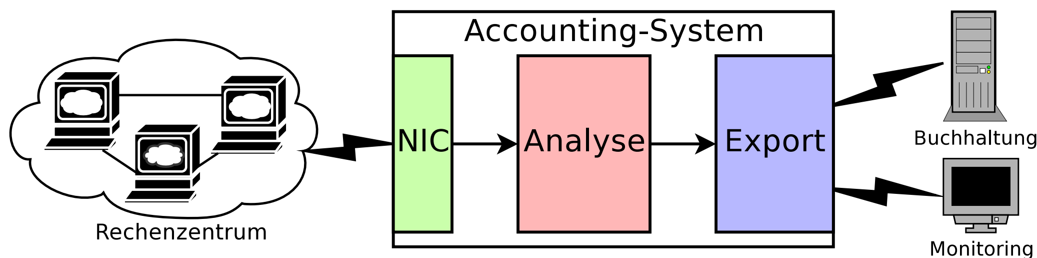

Abbildung 3.1: Datenfluss beim Accounting

Diese drei Phasen bilden drei voneinander fast unabhängige Module – es werden lediglich gemeinsame Strukturen zur Datenspeicherung benötigt. Das erlaubt es auch, die einzelnen Verarbeitungsschritte in eigene Module zu kapseln, und in Zukunft neue Module schnell und unkompliziert einzupflegen.

Da die Auswertung der Pakete den höchsten Aufwand erfordert, ist es am sinnvollsten, diese Aufgabe auf mehrere parallele Threads zu verteilen, und die Datenstrukturen entsprechend auszulegen, um Synchronisation zwischen den einzelnen Threads, aber auch mit der Aufnahme und der Ausgabe sicherzustellen.

3.2 Paket-Aufnahme

Zu jedem von der Netzwerkkarte aufgenommenen Paket müssen neben seinem Header auch einige Metadaten erfasst werden. Die vollständige Speicherung des Paketinhalts ist nicht nur unnötig, sondern auch ineffizient. Zur Kategorisierung von IPv4-Paketen mit VLAN-Tags genügen 38 Bytes (18 für den Ethernet- und 20 für den IP-Header), für IPv6 werden dagegen schon 58 Bytes benötigt. Soll eine weitere Aufteilung, z.B. nach TCP-Ports, notwendig werden, muss des Weiteren auch die Größe der möglichen IP-Zusatzheader in Betracht gezogen werden. Zum Paketheader kommen die den Zeitpunkt der Paketerfassung, die tatsächliche Länge des Pakets und die Anzahl der gespeicherten Bytes umfassenden Metadaten.

3.2.1 PCAP

Die einfachste und bequemste Art, diese Daten zu erhalten, bietet die Bibliothek PCAP, die entweder mit einem blockierenden Funktionsaufruf neu empfangene Pakete zurückliefert (blockierend heißt, dass die Funktion so lange wartet, bis auch ein Paket empfangen wurde), oder eine selbstdefinierte Funktion aufruft, wenn Pakete ankommen. Da die meisten Netzwerkanalyseprogramme auf dieser Bibliothek aufbauen, ist sie auf vielen Rechnern bereits vorhanden, und stellt eine gute Basis dar.

Allerdings ist der Aufwand zum Einlesen eines Pakets bei der libpcap so groß, dass sie sich bei über 70 Tausend Paketen pro Sekunde selbst auf einem High-End-Serversystem nicht mehr sinnvoll einsetzen lässt. Aber auch bei geringeren Datenmengen ist dieser Overhead unnötig und verlangsamt nur die Auswertung.

3.2.2 PF_RING

Eine Alternative, mit der das Ziel der passiven Paketaufnahme an einer eigens dafür vorgesehenen Netzwerkkarte – also genau das Anwendungsszenario – realisieren lässt, ist der PF_RING-Treiber. Damit werden die Paketdaten vom Kernel unmittelbar nach ihrem Empfang in einen Ringpuffer geschrieben. Dieser Ringpuffer wird dann in den Adressraum der Anwendung eingebunden und direkt ausgelesen. Der Aufbau der abgespeicherten Metadaten entspricht dem Format der libpcap, so dass bei der Auswertung keine größeren Anpassungen notwendig sind.

Da dieser Treiber nicht Bestandteil des Standard-Kernels ist, muss er manuell eingepatcht und der Kernel neu kompiliert werden. Obwohl das für ein dediziertes Accounting-System eine Selbstverständlichkeit ist, sollte die Anwendung auch ohne diese Manipulation lauffähig sein.

Es existiert ein Patch für die libpcap, mit dem die Bibliothek einen eventuell vorhandenen PF_RING-Puffer verwendet, jedoch ist damit der doppelte Aufwand des Transfers vom Kernel zur PCAP und von da aus zur Anwendung verbunden, was dem High-Performance-Anspruch entgegensteht.

Aus diesem Grund wurde die Anwendung mit zwei Paketempfangsmodulen ausgestattet. Wird ein mit PF_RING gepatchter Kernel festgestellt, greift die Anwendung darauf zurück – ansonsten verwendet sie die libpcap.

Sollte dies nicht alle Einsatzfälle abdecken, ist die Einbindung eines weiteren Moduls möglich. Die dafür vorgesehene Schnittstelle und die zu beachtenden Situationen werden in Kapitel 4.1 detailliert beschrieben.

3.3 Zwischenspeicherung

Zur Zwischenspeicherung der Paketdaten vor der Auswertung bietet sich eine verkettete Liste oder ein Ringpuffer an. Während eine Liste recht einfach zu handhaben ist, erzeugt die dafür nötige dynamische Speicherverwaltung einen hohen Mehraufwand. Außerdem muss man eine Begrenzung für den Fall der Überlastung einführen und diese durchsetzen.

Ein Ringpuffer dagegen belegt immer konstanten Speicher, und lässt sich bei empfangenen Paketen blockierfrei beschreiben, solange nur ein Thread Pakete einfügt. Da dies bei der Anwendung der Fall ist, wurde ein zentraler Ringpuffer als Paketspeicher eingesetzt. Bei Verwendung des PF_RING-Sockets greift die Anwendung dabei aus Performancegründen auf den kerneleigenen Ring zurück.

3.3.1 Ring-Synchronisation

Die Synchronisierung der lesenden Threads lässt sich dadurch erreichen, dass jeder Thread jeweils nicht überlappende Bereiche des Ringpuffers zur Abarbeitung bekommt. Dadurch lassen sich dedizierte Zwischenspeicher für jeden Thread vermeiden.

Im konkreten Fall werden die Ringpuffer-Zellen statisch auf die Analyse-Threads verteilt: bei N Threads kriegt ein Analysator jede Nte Zelle.

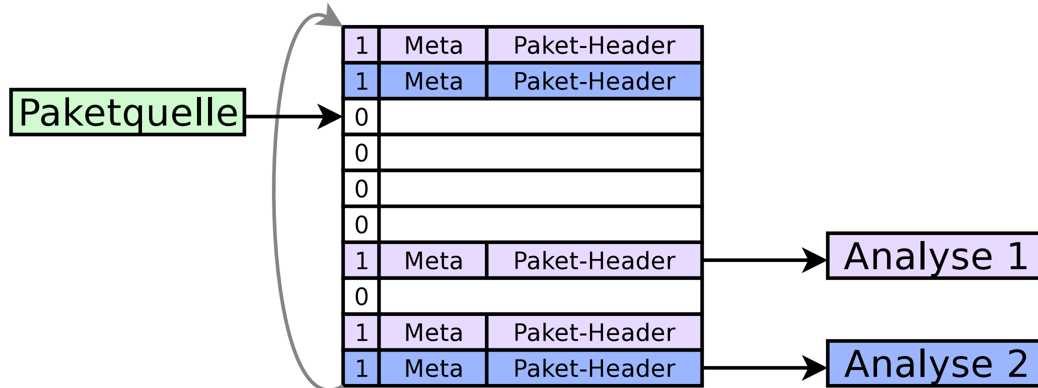

Abbildung 3.2: Ringpuffer mit zwei Analyse-Threads

Bei Verwendung des PF_RING-Treibers wird bereits vom Kernel ein Ringpuffer mit den Paketinhalten bereitgestellt, der dann sofort für die Auswertung benutzt werden kann. Der Aufbau dieses Rings wird auch für andere Paketquellen verwendet, da er alle relevanten Daten enthält.

Zusätzlich zu den oben beschriebenen Daten enthält jede Zelle des Puffers auch ein Status-Byte, das darüber entscheidet, ob der Inhalt der Zelle bearbeitet wurde oder nicht. Der schreibende Thread (bzw. der Kernel) setzt dieses Byte auf 1, und es wird von einem auswertenden Thread wieder auf 0 gesetzt, wenn das Paket bearbeitet wurde.

Wenn die nächste zu schreibende Zelle eine 1 im Statusfeld enthält, ist der Ringpuffer voll und neue Pakete werden unbearbeitet verworfen. Obwohl das keine für den Normalbetrieb wünschenswerte Eigenschaft ist, führt das bei einer Überlastung des Systems zu einer selbständigen Regelung des maximalen Durchsatzes – da Pakete, für deren Bearbeitung die Rechenleistung nicht mehr ausreicht, ganz am Anfang der Bearbeitung verworfen werden, können andere Pakete vollständig abgearbeitet werden.

Das Statusfeld dient aber auch der Synchronisation in die umgekehrte Richtung – daran kann ein Anaylse-Thread erkennen, ob sich neue Daten im Ringpuffer befinden, und diese dann auswerten.

Liegen in dem Ringpuffer dagegen keine Daten vor, legt sich der Analyse-Thread kurz auf Lauer und wartet auf neue. Kommen dann Datenpakete an, nehmen die Threads ihre Analysearbeit wieder auf. Obwohl durch dieses Polling eine geringfügige Mehrbelastung des Systems entsteht, wenn keine Pakete vorliegen, hat es sich bei praktischen Versuchen als die effektivste Methode herausgestellt, auf einem stark ausgelasteten System alle Pakete zu verarbeiten.

Es kann wie in der Abbildung dargestellt auch vorkommen, dass einer der Threads die anderen beim Abarbeiten des Puffers überholt. Das ist nicht weiter problematisch, da die Threads vom Scheduler gleich behandelt werden, so lange sie Arbeit haben – was ihr Auseinanderdriften gering hält – und wieder zusammenkommen, sobald sie alle Daten im Ringpuffer abgearbeitet haben.

3.4 Analyse

Das Ziel der Analyse besteht darin, je nach Anwendererfordernis die Pakete zu einzelnen Verbindungen, IP-Adressen oder Rechnern zuzuordnen, und für diese die Anzahl der Pakete und Bytes mitzuzählen.

Eventuell müssen dabei zuerst in den Ethernet-Paketen eingebettete VLAN-Tags ausgewertet oder verworfen werden. Diese Tags sind Markierungen, die auf Switch-Ebene die Zugehörigkeit zu einem bestimmten logischen Netz anzeigen, und zur besseren Gliederung der Topologie in großen Netzwerken eingesetzt werden.

3.4.1 Kundenerkennung

Für die Rechnungsstellung genügt es, den Datenverkehr nach den Kunden-IPs aufzuschlüsseln. Da aber in jedem Paket zwei IP-Adressen (Absender und Empfänger) vorkommen, muss eine Überprüfung stattfinden, welche davon einem Kunden zuzordnen ist. Dazu wird im Programm eine Liste von Netzwerkadresse- und Netzwerkmaske-Paaren vorgehalten. Eine IP-Adresse wird beim Vergleich erst mit der Maske AND-verknüpft und dann mit der Netzwerkadresse verglichen. Stimmen die Werte überein, so ist die Adresse Teil von dem Netz und gehört damit einem Kunden. Wenn bei keinem der eingetragenen Netzwerke eine Übereinstimmung gefunden wird, ist es dagegen keine Kunden-IP. Um den Vergleich effizienter zu gestalten, liegen die Netzwerkangaben in der Netzwerkbyteanordnung vor, so wie die IP-Adresse im Paket - damit ist ein Vergleich mit zwei Binäroperationen abgeschlossen.

Wenn eine der beiden IP-Adressen aus dem Paket einem Kunden gehört,

muss das Paket ihm als ausgehend respektive ankommend angerechnet werden. Gehören beide Adressen zu Kunden, kann das Paket optional verworfen werden, um Datenverkehr innerhalb des Rechenzentrums nicht in Rechnung zu stellen.

Treten dagegen Pakete auf, die weder von einer bekannten IP kommen, noch an eine gerichtet sind, ist davon auszugehen, dass entweder ein Rechner falsch konfiguriert wurde, oder dass ein Angriff mit gefälschten Absenderadressen stattfindet. Hier werden die Paketdaten direkt ausgegeben, damit anhand der MAC-Adresse der sendende Rechner bestimmt werden kann.

3.4.2 Anordnung der Zähler

Hat man das Paket vorklassifiziert, gilt es als nächstes möglichst schnell die Zähler dafür zu finden und zu aktualisieren. Der naive Ansatz besteht darin, einen Baum mit einer Breite von 256 zu wählen, so dass man für jedes Byte der IP-Adresse eine Ebene weiterkommt, und nach vier Zugriffen (bzw. nach sechs für IP und Port) an dem Blatt mit dem gesuchten Zähler angelangt ist. Will man jedoch andere Unterscheidungskriterien wie die MAC-Adresse integrieren, oder IPv6-Pakete klassifizieren, wo alleine die Adresse 16 Bytes belegt, wird das Verfahren unpraktisch.

Die Verwendung einer linearen Liste verbietet sich von selbst, da dort für jedes Paket ein Suchaufwand von $O(N)$ nötig ist. Ausgeglichene Bäume jeglicher Arten und sortiere Felder bieten zwar im Durchschnittsfall $O(logN)$, jedoch ist auch das nicht optimal (Der Wert N beschreibt dabei die Anzahl der bereits vorhandenen Zähler).

Gut geeignet ist dagegen eine Hashtable hinreichender Größe, wobei der Hashwert jeweils über die unterscheidungsrelevanten Teile des Datenpakets, also z.B. das Protokoll, die Kunden-IP und die MAC-Adresse gebildet wird.

3.4.3 Hash-Tabellen

Die eineindeutige Abbildung eines so großen Werteraums auf eine Tabelle, die in den Arbeitsspeicher gängiger Systeme passen soll, ist nicht möglich. Daher muss mit Kollisionen in der Hashtabelle gerechnet werden. Deren Auflösung erfolgt dadurch, dass Einträge mit gleichem Hashwert verkettet gespeichert werden, und zu jedem Eintrag sein eineindeutiger Schlüssel abgelegt wird.

Sucht man einen Eintrag, so muss man zunächst den Hashwert bilden, um die Stelle in der Tabelle auszumachen. Dann vergleicht man den Schlüssel des aktuellen Tabelleneintrags mit dem gesuchten Schlüssel, und hangelt sich so lange durch die Liste, bis eine Übereinstimmung gefunden wird.

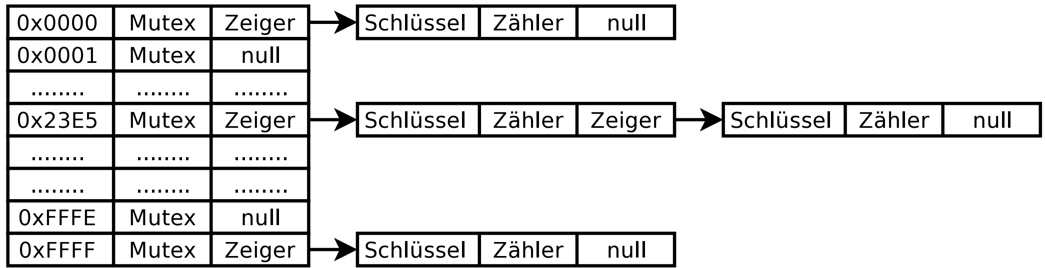

Abbildung 3.3: Hash-Tabelle für Paketzähler (64K Einträge)

Wird beim Durchhangeln kein Eintrag gefunden, so wird noch im selben Durchgang einer erzeugt. Da dabei die Struktur der Liste verändert wird, und andere Threads gerade daruf zugreifen könnten, muss der gesamte Zugriff durch einen Mutex gekapselt werden. Würde aber die gesamte Hash-Tabelle nur über einen zentralen Mutex zugänglich sein, könnte immer nur ein Thread auf ein Mal seine Arbeit machen. Daher existiert für jede einzelne Hashzelle ein eigener Mutex, so dass die Threads parallel unterschiedliche Pakete auswerten und in die Tabelle einfügen können.

Solange es keine Kollisionen gibt, hat das Wiederfinden eines Eintrags die Komplexität $O(1)$, auch bei nur wenigen Schlüsseln mit übereinstimmendem Hash wird die Suche nicht wesentlich verlangsamt. Für eine angemessen dimensionierte Hashtabelle ist das der Normalfall.

Gelingt es jedoch einem Angreifer, gezielt Pakete einzuschleusen, die bei unterschiedlichen Schlüsseln auf den selben Hashwert abgebildet werden, steigt die Komplexität auf $O(N)$ für die Anzahl der bereits gespeicherten Angreiferpakete. Damit wird Rechenzeit gebündelt und es ist nicht mehr möglich, alle Pakete auszuwerten. Dieser Angriff wurde in [7] vorgestellt, und auch erfolgreich gegen den IP-Stack im Linux-Kernel angewandt[8].

Ein Angriff auf das Accounting-System erscheint gar nicht so unwahrscheinlich, da dadurch zum einen die Erkennung anderer Angriffe erschwert wird, und zum Anderen der Angreifer die Möglichkeit erhalten würde, auf Kosten des Betreibers Daten zu übertragen, da sie nicht gezählt und ihm in Rechnung gestellt werden können.

Eine Lösung dafür würde in der Verwendung einer kryptografisch sicheren Hash-Funktion bestehen, jedoch sind solche zu langsam für den gegebenen Anwendungsfall. Es ist auch möglich und ausreichend, eine Hash-Funktion zu verwenden, bei der Kollisionen nicht trivial erzeugbar sind, und die mit einem Zufallswert initialisiert wird. Daher wurde in der Implementierung der Hash-Algorithmus von Bob Jenkins[6] eingesetzt, der als Public Domain verfügbar ist und auch im Linux-Kernel zur Lösung des Denial of Service-Problems

verwendet wurde. Dieser hat auch den Vorteil, extrem schnell zu sein, da er nur auf Bit-Verschiebungen und -Verknüpfungen zurückgreift.

Dennoch ist eine hinreichende Dimensionierung der Hashtabelle wichtig. Als Faustregel gilt, dass die Tabelle Platz für mindestens doppelt so viele Einträge haben muss, wie dafür erwartet werden. Um Divisionsoperationen einzusparen muss die Größe des weiteren eine Zweier-Potenz sein. Sollen z.B. nur IP-basierte Zähler für ein Netz mit 3500 Systemen verwendet werden, reichen 8192 Einträge. Geht es dagegen auch darum, nach TCP/UDP-Ports aufzuschlüsseln, sollten es mindestens 128K Einträge sein.

Aus Performancegründen ist eine dynamische Vergrößerung der Tabelle im laufenden Betrieb nicht zweckmäßig, da alle Einträge umsortiert werden müssten. Es wäre dagegen möglich, nach einem Exportdurchlauf eine neue Hashtabelle mit mehr Platz zu erzeugen, dies wird aber im Haupteinsatzszenario nicht benötigt. Außerdem spielt die Größe der Tabelle nur eine geringe Rolle bei der Analyse-Performance, so dass eine Überdimensionierung keine Probleme mit sich bringt.

3.4.4 Zähler und Synchronisation

Ist der passende Datensatz gefunden worden, müssen die darin enthaltenen Zähler für die Anzahl der Pakete und der Bytes erhöht werden. Beim Byte-Zähler ist dabei die Größe abzüglich des Ethernet-Headers einzutragen, da dieser nur im LAN eine Rolle spielt. Außerdem ist zu beachten, dass die Zähler mit 64 Bit ausgelegt werden, da bei 32 Bit ein Überlauf nach 4 GByte erfolgt, und diese Datenmenge schon auf einem 100-MBit-Netzwerk nach 6 Minuten zu erreichen ist.

Diese Zähler lassen sich aber nicht mehr auf 32-Bit-Rechnern atomar erhöhen, so dass auch diese Aufgabe in einem kritischen Bereich zu erfolgen hat. Da das im Verhältnis zu der Suche in der Hashtabelle nur verhältnismäßig wenig Zeit beansprucht, kann auch der Hash-Mutex dafür mitverwendet werden, was weiteren Overhead einspart.

3.4.5 Mehrere Hash-Tabellen

Es erscheint auf den ersten Blick sinnvoll, jeden Analyse-Thread mit einer eigenen Hash-Table zu versehen, um so die Ausgabe der einzelnen Threads voneinander zu entkoppeln. Diese Einzeltabellen müssen aber bei jedem Exportvorgang zusammengefügt werden, um Duplikate zu erkennen. Außerdem erhöhen sie den Speicherbedarf des Programms und verursachen mehr kostspielige Transfers zwischen CPU-Cache und Arbeitsspeicher.

Auf der anderen Seite benötigen thread-lokale Hashtabellen keinerlei Synchronisationsmechanismen beim normalen Zugriff, da kein anderer Thread darauf zugreifen kann.

Um die Vor- und Nachteile der Alternativen im praktischen Einsatz prüfen zu können, wurden beide in der Anwendung implementiert und per Präprozessor-Direktive umschaltbar gemacht.

3.5 Daten-Export

Zu jedem Exportzeitpunkt wird die Hashtable gesichert, und den Analyse-Threads eine neue vorgesetzt. Hat jeder Thread eine eigene, werden alle Tabellen zu einer zusammengeführt. Der Inhalt dieser wird dann vom Ausgabemodul sequentiell abgearbeitet und in einer von diesem Modul bestimmten Form exportiert.

Das Zusammenfügen mehrerer Tabellen läuft dabei so ab, dass jede der Einzeltabellen durchgegangen wird und ihre Einträge in die neue Hash-Tabelle übernommen werden.

Da die Analyse-Threads für jedes Paket auf die Hashtabelle zugreifen, kann diese nicht einfach ersetzt werden. Daher wird die gesamte Arbeit mit ihr als kritischer Bereich definiert, in dem sich entweder der auswertende Thread oder nur der die Tabelle ersetzende Prozess befinden darf. Obwohl dies eine atomare Operation (nämlich das Überschreiben eines Zeigers mit einem anderen) ist, muss sichergestellt werden, dass keine Arbeit mehr an der alten Hashtable stattfindet.

Ansonsten kann es passieren, dass ein auswertender Thread den Zeiger auf die gerade noch gültige Hash-Tabelle kopiert, und an dieser Schreiboperationen durchführt, während diese bereits vom Exportierer ausgegeben und ihr Speicherplatz freigegeben wird. Das würde im besten Fall Datenleichen im Speicher hinterlassen und im schlimmsten Fall die Anwendung zum Absturz bringen. Zur Synchronisierung wird eine Semaphore verwendet, die es den Analyse-Threads erlaubt, mit der Hashtabelle zu arbeiten, und die die Threads beim Austausch der Tabelle kurz blockiert.

Das implementierte Ausgabemodul erzeugt eine Klartextdatei mit einer Auflistung aller IP-Adressen und den zugehörigen MAC-Adressen und Paket- und Byte-Zählern. Andere Module, z.B. zum Einspeisen der Daten in eine relationale Datenbank, sind zwar vorgesehen, wurden aber nicht im Rahmen der Aufgabe implementiert.

Kapitel 4

Implementierung

Nachdem das Konzept und der Aufbau der Anwendung im vorangegangenen Kapitel vorgestellt wurden, geht es hier um konkrete Details wie die Auslegung der Schnittstellen für Module, und die dafür notwendigen Strukturen, sowie um die innere Funktionsweise und für einen ungestörten Ablauf nötige Bedingungen.

Die Anwendung wurde in C entwickelt, um möglichst viel Kontrolle über alle performance-kritischen Bereiche und über die Datenstrukturen zu haben. Zur Konfiguration der Übersetzung dienen die automake/autoconf-Tools, und es wurde der GCC-Compiler in den Versionen 3.2.3 und 3.4.3 verwendet.

Entsprechend den Ansprüchen des Programms, Datenverkehr über Gigabit-Glasfaser auszuwerten, wurde als Titel „Faster than Light IP Accounting" (flipacc) gewählt.

Die Implementierung richtete sich an den in Kapitel 3 vorgestellten Algorithmen und Methoden. Der Anfang und das Ende der Verarbeitungskette wurden dabei modular implementiert, und die dafür vorgesehenen Schnittstellen werden in den nächsten zwei Abschnitten vorgestellt. Im Anschluss wird dann auf den Programmablauf dazwischen eingegangen.

4.1 Schnittstelle zum Packet-Capturing

Die Hauptaufgabe beim Packet Capturing besteht darin, Pakete vom Treiber der Netzwerkkarte entgegenzunehmen und der Anwendung zur Verfügung zu stellen. Ein Capturing-Modul durchlebt drei Phasen – Initialisierung, Betrieb und Beendigung.

4.1.1 Initialisierung eines Capturing-Moduls

Die Initialisierung eines Capturing-Moduls erfolgt über den Aufruf der von diesem Modul bereitgestellten `cpt_*_open(...)`-Methode. Diese bekommt in drei Parametern den Namen der Netzwerkkarte, die gewünschte Größe der zu speichernden Paketfragmente und ein Flag, ob die Netzwerkkarte in den Promiscuous Mode gebracht werden soll, in dem sie alle Pakete empfängt.

Die Initialisierungsroutine erzeugt eine neue Struktur vom Typ `struct capturer` (Listing 4.1) und belegt diese vollständig mit Daten. Als Ergebnis liefert sie einen Zeiger darauf zurück, oder `NULL` im Fehlerfall.

```
typedef struct capturer {
	void		*priv;
	struct {
		int	ringsize;  /* number of elements */
		int	snaplen;   /* data bytes per slot */
		int	slotsize;  /* snaplen + hdr + padding */
		uint8_t *ring;
	} ring;
	uint64_t	pkt_total, pkt_dropped;
	int		(*collect)(struct cpt_control*);
	int		(*expect)(struct capturer*);
	int		(*close)(struct capturer*);
};
```

Listing 4.1: `struct capturer` - Beschreibung eines Packet Capturer

Das erste Element von `struct capturer`, der `priv`-Zeiger, ist für die Verwaltung der Capturer-eigenen Daten gedacht. Der von ihm referenzierte Inhalt soll den `collect()`- und `close()`-Funktionen später genügen, um Pakete erfolgreich aufzunehmen und die Paketquelle zu deinitialisieren.

Auf den `priv`-Zeiger folgen die Meta-Daten für den Ringpuffer. Sie dienen in erster Linie der Flexibilität der Quellenauswahl – so kann hier ein bereits existierender Ringpuffer direkt eingebunden oder ein eigener erzeugt werden. `ringsize` beschreibt die Anzahl der Zellen im Puffer, `snaplen` die maximale Größe für ein Paketfragment und `slotsize` die Größe einer Zelle. Diese ergibt sich aus der Summe von `snaplen`, der Größe der Metadaten für eine Ringzelle und einem eventuellen Padding.

Die Gesamtgröße des Rings ergibt sich dabei als `ringsize*slotsize`, und ein Zeiger auf den Ring muss sich in `ring` befinden.

Darauf folgen zwei Zähler für die Anzahl der empfangenen und der vom Capturer verworfenen Pakete. Diese sollten beim Initialisieren zurückgesetzt werden, und spätestens beim `close()`-Aufruf mit Werten belegt werden.

Beendet wird die capturer-Struktur mit drei Funktionszeigern, die bei der Initialisierung auf die entsprechenden Funktionen des Capturing-Moduls gesetzt werden müssen. Ihre Aufgaben werden in den folgenden Abschnitten erklärt.

```
struct capturer *cpt = cpt_pcap_open("eth0", 64, 1);
if (cpt == NULL) {
        perror("cpt_pcap_open()");
        exit(1);
}
```

Listing 4.2: Beispiel der Initialisierung eines Capturing-Moduls

4.1.2 Sammlung der Daten - collect()

Nachdem das Capturing-Modul erfolgreich über seine open()-Funktion initialisiert wurde, legt die Anwendung eine struct cpt_control an, die einen Zeiger auf den Capturer und ein running-Flag enthält. Danach startet sie einen neuen Thread und ruft darin cpt->collect() mit der cpt_control-Struktur als Parameter auf.

Von dieser Funktion wird erwartet, neue Datenpakete zu empfangen und ihren Inhalt sowie die Metadaten in den Ringpuffer zu schreiben, bis das running-Flag auf 0 zurückgesetzt wird.

Dabei müssen die Zellen des Ringpuffers den folgenden Aufbau haben:

```
typedef struct cpt_ring_slot {
        uint8_t       magic; /* == SLOT_MAGIC == 0x88 */
        uint8_t       slot_state;
        struct cpt_packet data;
} __attribute__((packed));

typedef struct cpt_packet {
        struct pcap_pkthdr meta;
        uint8_t       data[];
};
```

Listing 4.3: struct cpt_ring_slot - Aufbau einer Ringzelle

Das Attribut packed beschreibt, dass kein Padding eingefügt werden darf, um den Zugriff auf einzelne Elemente zu beschleunigen. Obwohl dies einen negativen Einfluss auf die Geschwindigkeit haben kann, ergibt es sich hier aus der Notwendigkeit, mit der Datenstruktur des PF_RING-Treibers binärkompatibel zu sein.

Das magic-Byte hat immer den Inhalt 0x88 und dient als Indikator für den Anfang einer Ringzelle. Das darauffolgende slot_state-Feld enthält eine 0, falls die Zelle leer ist, oder eine 1, sofern sie Daten enthält. Dieses Flag wird vom schreibenden Thread erst gesetzt, nachdem er das Paket in die Zelle übertragen hat, und vom lesenden erst dann gelöscht, wenn die Daten verarbeitet wurden. Trifft der schreibende Thread beim beschreiben des Ringpuffers auf eine Zelle, wo das Flag noch gesetzt ist, ist der Puffer voll und der Schreiber muss so lange Pakete verwerfen, bis die Zelle als frei markiert wird.

Danach kommt eine Struktur, die die Meta- und die eigentlichen Paketdaten zusammenfasst. Erstere bestehen aus einem struct pcap_pkthdr (Listing 4.4), in dem der Zeitstempel des Empfangs und die gelesene sowie die tatsächliche Größe des Pakets eingetragen sind. Das nach dieser Struktur folgende Byte-Feld data[] ist ein flexibles Array, dessen Größe zur Laufzeit bestimmt wird. Im konkreten Fall belegt es den Platz bis zum Anfang der nächsten Ringpufferzelle. Die Anzahl der zur Verfügung stehenden Bytes wurde dabei schon von der Initialisierungsfunktion festgelegt und in capturer.ring.snaplen eingetragen.

```
119 struct pcap_pkthdr {
120         struct timeval ts;     /* time stamp */
121         bpf_u_int32 caplen;    /* length of portion present */
122         bpf_u_int32 len;       /* length of packet (off wire) */
123 };
```

Listing 4.4: struct pcap_pkthdr - Metadaten für ein empfangenes Paket

4.1.3 Synchronisation - expect()

Sobald einer der Analyse-Threads keine Pakete mehr in seinem Ringpuffer-Bereich vorfindet, ruft er die expect()-Funktion des Capturing-Moduls auf.

Die Aufgabe der Funktion besteht darin, darauf zu warten, dass neue Daten im Ringpuffer vorliegen oder dass das Programm abgebrochen wurde. Dafür erhält sie als Parameter zum einen die cpt_control-Struktur der Anwendung und zum anderen die aktuelle Position im Ringpuffer.

So kann ein Capturing-Modul intern beliebige Methoden der Synchronisation verwenden, um neue Daten anzukündigen, und dies bleibt vor der Anwendung selbst verborgen.

Bei Vorliegen von Abbruchkriterien soll die Funktion -1 als Rückgabewert liefern, wenn dagegen Daten im Ringpuffer an der angeforderten Position vorliege, soll sie das mit einer 0 quittieren.

Zu beachten ist, dass diese Methode von mehreren Threads gleichzeitig aufgerufen werden kann, und man entsprechende Maßnahmen zur Vermeidung von internen Konflikten ergreifen muss.

4.1.4 Aufräumen - `close()`

Bei Beendigung der Anwendung werden zuerst der Capturer-Thread in dem `collect()` läuft und die Analyse-Threads beendet. Danach wird die `close()`-Funktion des Capturing-Moduls aufgerufen, deren Aufgabe darin besteht, eventuell offene Dateideskriptoren zu schließen und die internen Strukturen freizugeben.

4.2 Schnittstellen zum Datenexport

In bestimmten Intervallen muss die Anwendung die bis dahin angefallenen Daten ausgeben. Zum einen wird dadurch die interne Hashtabelle von bereits veralteten Zählern befreit, zum anderen dienen die Ausgaben der kontinuierlichen Überwachung des Datenaufkommens.

Für jeden Exportvorgang wird dabei die bisher verwendete Hashtable von den Analyse-Threads entkoppelt und an das eingestellte Exportmodul übergeben. Damit die Auswertung weiter laufen kann, wird dabei sofort eine neue Tabelle angelegt, die den Platz der alten einnimmt.

Die Verwendung thread-lokaler Hashtabellen ist für das Exportmodul dabei transparent, da sich die Anwendung darum kümmert, die lokalen Tabellen zu einer gemeinsamen zusammenzufügen, die dann an das Exportmodul übergeben wird.

Die Exportfunktion bekommt daher als Parameter eine `struct hashtable` mit den Metadaten der gemeinsamen Hash-Tabelle und einem Zeiger auf die eigentlichen Daten:

```
struct hashtable {
        uint32_t                iv;
        uint32_t                bits;
        struct anl_counter      **table;
        [...]
};
```

Listing 4.5: `struct hashtable` - Metadaten einer Hash-Tabelle

Das erste Feld enthält den Initialisierungsvektor für die Hashfunktion. Dieser wird wie in Abschnitt 3.4.3 erläutert zur Laufzeit auf einen Zufallswert gesetzt, um Kollisionsangriffe zu erschweren. Da der Wert nur für die

Zuordnung von Schlüsseln auf Hashwerte wichtig ist, ist er für die Auswertung nicht relevant.

Das zweite Element der Struktur, `bits`, speichert die Anzahl der Bits, die bei einem Hashwert relevant sind. Daraus ergibt sich dann auch die Größe der Hashtabelle, da diese $2^{\texttt{bits}}$ Einträge umfassen muss. Zur Umrechnung sollten die beiden Makros `hashsize(bits)` und `hashmask(bits)` verwendet werden, von denen das erste die Anzahl der Elemente und das zweite die dazu passende Bitmaske für AND-Verknüpfungen zurückliefern.

Das dritte und wichtigste Feld ist der Zeiger `table`, der auf ein Array mit `hashsize(bits)` Einträgen vom Typ `struct anl_counter*` (also auch Zeiger) verweist.

```
#define MAXKEYBUF 56
typedef struct anl_counter {
        int             keylen;
        union {
                uint8_t         key[MAXKEYBUF];
                int             protocol;
        } key;
        struct anl_counter *next;
        uint64_t        packets_in;
        uint64_t        bytes_in;
        uint64_t        packets_out;
        uint64_t        bytes_out;
};
```

Listing 4.6: `struct anl_counter` - Aufbau Paketzähler

Jeder dieser Zeiger ist der Anfang einer verketteten Liste von Zählern, die für die Auswertung durchgegangen werden muss. Der Aufbau der Zählerstruktur ist in Listing 4.6 dokumentiert.

Die ersten beiden Felder beschreiben den Hash-Schlüssel, der für diesen Eintrag verwendet wurde – `keylen` enthält die Anzahl der relevanten Bytes von `key`. Dabei ist `key` als Union von einem Bytefeld und dem Integer-Wert `protocol`, der den Typ der abgespeicherten Daten beschreibt, ausgelegt.

```
struct hash_ip4_key {
        int             protocol;       /* HASH_PROTO_IP4 */
        uint32_t        addr;       /* network byte order */
        uint8_t         ether[ETH_ALEN];
} __attribute__((packed));
```

Listing 4.7: `struct hash_ip4_key` - IPv4-Protokolldaten

```
struct hash_ip6_key {
	int		protocol;	/* HASH_PROTO_IP6 */
	struct in6_addr addr;	/* IPv6 address */
	uint8_t		ether[ETH_ALEN];
} __attribute__((packed));
```

Listing 4.8: `struct hash_ip6_key` - IPv6-Protokolldaten

Als Protokolle sind bisher Ethernet, IPv4 (Listing 4.7) und IPv6 (4.8) definiert (Die konstanten findet man in `enum hash_protocol`). Die protokollspezifischen Daten befinden sich in den weiteren Bytes des Hashkeys. Das Export-Modul muss alle relevanten Protokolle erkennen und korrekt ausgeben können.

Leider ist es nicht ohne deutlichen Aufwand (wie z.B. der Verwendung einer Datenbeschreibungssprache wie XML oder ASN.1) möglich, zwischen nachrüstbaren Protokollauswertern und nachrüstbaren Exportfiltern zu vermitteln. Daher muss in jedem Exportmodul jedes bekannte Protokoll ausgegeben werden können, und das Auftreffen von unbekannten Protokollen sollte durch eine Warnung quittiert werden.

In `next` befindet sich der Zeiger auf das nächste Listenelement mit gleichem Hashkey, und darauf folgen die eigentlich relevanten Variablen - die vier Zähler für Pakete und Bytes ankommend und ausgehend.

Aus den über den Key gelegten `hash_*_key`-Strukturen lassen sich somit die Daten des Kunden entnehmen, und zusammen mit den zugehörigen Zählerständen protokollieren. Die Variablen sind `packed`, da die Hash-Checksumme über die gesamte Datenstruktur gebildet wird, und eventuell uninitialisierte Padding-Bytes den Hash-Wert verfälschen würden.

4.3 Weitere Besonderheiten

In diesem Abschnitt sollen einige interne Strukturen des Programms näher erklärt werden, die zwar nicht unmittelbar für Erweiterungen wichtig sind, aber dabei helfen können, ein Verständnis für die Abläufe zu entwickeln.

4.3.1 Analyse-Ablauf

Jeder Analyse-Thread führt die Funktion `anl_run()` aus, deren Aufgabe darin besteht, in einer Schleife auf neue Pakete zu warten und diese auszuwerten. Die dafür notwendigen Metadaten werden dem Thread über die im folgenden Abschnitt erläuterte `anl_control`-Struktur übergeben.

Die Funktion läuft so lange, wie die `running`-Variable gesetzt ist, und schaut dabei immer, ob ein Datenpaket an der Leseposition des Threads vorliegt. Ist das nicht der Fall, wird die `expect()`-Funktion des Capturers aufgerufen, die erst zurückkehrt, wenn ein neues Paket in der aktuellen Ringzelle liegt.

Liegt ein Paket zum verarbeiten vor, wird es an `anl_ether()` übergeben, wo eine Überprüfung auf VLAN-Tags und eine Weitervermittlung an die für das gefundene höhere Protokoll zuständige Funktion (also `anl_ipv4()` oder `anl_ipv6()`) stattfindet.

Letztere sind dafür zuständig, aus dem Paket die Kunden-Rechner-Adresse herzuleiten und den sich daraus ergebenden Hashwert zu bilden. Über den Hashwert passen sie dann die dem Rechner entsprechenden Byte- und Paketzähler an, und geben die Kontrolle an `anl_run()` zurück.

Dort werden dann die thread-lokalen Zähler angepasst und der Leseindex um die Gesamtanzahl der Threads erhöht. Da die Indizes der einzelnen Threads zueinander versetzt sind, wird auf diese Weise mit geringem Aufwand sichergestellt, dass keine Überlappungen der abzuarbeitenden Ringbereiche auftreten.

4.3.2 Kontrolle der Analyse-Threads

Zur Mitteilung aller wichtigen Daten an einen Analyse-Thread dient die Struktur `struct anl_control`. Sie wird bei der Thread-Initialisierung als Parameter übergeben und ist damit die beste Möglichkeit, den Thread mit allen wichtigen Informationen zu versehen.

```
1 typedef struct anl_control {
2         pthread_t      thread;
3         int            thread_id;
4         int            threadcount;
5         struct cpt_control *cc;
6         struct hashtable *hashtable;
7         uint64_t       total_packets;
8         uint64_t       total_bytes;
9 } struct_anl_control;
```

Listing 4.9: `struct anl_control` - Kontrollstruktur für Analyse-Threads

Die Struktur enthält neben den implementierungsspezifischen Thread-Daten auch die logische Nummer des Threads, die Gesamtanzahl der Auswerter und einen Verweis auf die Kontrollstruktur für das Packet-Capturing.

Der Zeiger auf den Capturer wird benötigt, um auf den Ringpuffer und die `running`-Variable zugreifen zu können.

Das wichtigste Element der Struktur ist ein Zeiger auf die Hashtabelle – diese nimmt die Ergebnisse der Arbeit auf, und wird in regelmäßigen Intervallen „geleert", also durch eine neue ersetzt.

Schließlich enthält die Struktur zu statistischen Zwecken Zähler für die Anzahl der von diesem Thread verarbeiteten Bytes und Pakete.

Das Hauptprogramm erzeugt für jeden Thread eine solche Struktur, und lässt diese bis nach der Terminierung der Threads per `pthread_cancel` / `pthread_join` zum Programmende bestehen.

4.3.3 VLAN-Erkennung

Ethernet-Pakete können eine optionale Erweiterung enthalten, in der das virtuelle LAN, zu dem sie gehören, verzeichnet ist. Solche Pakete sind 4 Bytes größer, und der tatsächliche Pakettyp ist im Erweiterungsheader zu finden, während an seiner eigentlichen Stelle `0x8100` als Kennzeichnung auftritt. Wertet man Daten an einem Switch aus, der Pakete auf seinem Uplink tagged, also mit VLAN-Kennungen ausstattet, so ergibt sich ein gemischtes Bild – während Pakete in die eine Richtung mit Tag auf den Mirror-Port ausgegeben werden, kommen die Pakete der Gegenrichtung ohne die Markierung. Die sinnvollste Vorgehensweise besteht daher darin, die Tags beim Auswerten zu überspringen.

Um das zu realisieren, übergibt die Analyseroutine für Ethernet-Frames den für das jeweils vorgefundene höhere Protokoll zuständigen Funktionen einen Offset im Datenpaket, der ohne VLAN bei 14, der normalen Ethernet-Header-Größe liegt, und für Frames mit VLAN entsprechend auf 18 gesetzt ist.

Obwohl es theoretisch möglich ist, mehrere VLAN-Ebenen ineinander zu verschachteln, wurde die dafür nötige Unterstützung explizit außen vor gelassen, da es ansonsten möglich wäre, Ethernet-Pakete zu konstruieren, die lediglich aus rekursiv aufgebauten VLAN-Tags und keinem eigentlichen Inhalt bestehen, um damit die auswertende Anwendung zusätzlich zu belasten.

Kapitel 5

Evaluierung

In diesem Kapitel wird vorgestellt, welche Resultate mit der implementierten Anwendung erreichbar sind und dokumentiert, wie diese Resultate im Testfall erreicht wurden.

Die praktische Einsatzfähigkeit des Programms wurde dabei in zwei Szenarien gemessen. Im ersten, dem Tuningszenario, ging es darum, bei konstanten Versuchsbedingungen das Verhalten des Programms in Abhängigkeit von unterschiedlichen Faktoren zu betrachten und zu optimieren. Dies erforderte konstante Messbedingungen und erfolgte daher mit synthetisch generierten Paketdaten.

Das zweite Szenario bestand darin, die Anwendung mit möglichst realistischen Daten zu testen und zu vergleichen, wie sie im Praxiseinsatz skaliert und auch gegenüber IPAudit abschneidet. Dazu wurde im Rechenzentrum ein schnelles Dual-Xeon-System für die Messungen bereitgestellt und über eine Gigabit-Netzwerkkarte an den Core-Router angeschlossen, der darauf alle aus seinem Internet-Upstream anfallenden Daten gespiegelt hat.

5.1 Tuningphase

5.1.1 Versuchsaufbau

Das verwendete Messsystem enthielt zwei Celeron-Prozessoren mit Mendocino-Kern und 500 MHz. Über den 440BX-Chipsatz von Intel wurden 1GB PC66-RAM und eine Intel 82540EM Gigabit-Ethernet-Karte (PCI, 32 Bit, 33 MHz) für das Capturing angebunden. Die Steuerung erfolgte über ssh, das über eine zusätzliche Netzwerkkarte lief.

Auf diesem Rechner lief ein Debian 3.1 testing mit selbst kompiliertem Linux-Kernel 2.6.10 und dem PF_RING-Patch.

Um kontrollierte Bedingungen zu erzeugen, wurde ein synthetischer Datenstrom an die Gigabit-Karte ausgegeben. Die Paketerzeugung erfolgte dabei durch das Linux-Modul pktgen. Der erzeugende Rechner war ein Pentium-M-1300 mit einer Broadcom BCM4401 Fast-Ethernet-Karte, die über ein Crosslink-Kabel mit dem Messsystem verbunden war.

Die erzeugten Pakete hatten dabei eine Größe von 60 Bytes auf dem Medium, also inclusive dem Ethernet-Header. Es handelte sich um UDP-Pakete von einer einzelnen Adresse an zufällig variierende Zieladressen aus einem /17-Netz, also insgesamt um 32768 unterschiedliche Verbindungen, für die eine Zuordnung erfolgen musste.

Der pktgen-Treiber erzeugte dabei ca. 120 Tausend Pakete pro Sekunde und lastete das Netz damit mit ca. 60 MBit/s aus. Durch die sehr geringe Paketgröße konnte er die für ein übliches GBit-Netzwerk erwartete Paketmenge simulieren.

5.1.2 Messergebnisse

Bei den ersten Versuchen wurde festgestellt, dass der Ring-Puffer von PF_RING in der Standardeinstellung mit 8000 Zellen sehr schnell überläuft. Daher wurden die Parameter für das Modul entsprechend angepasst:

```
modprobe ring bucket_len=61 num_slots=65536
```

Listing 5.1: Initialisierung des PF_RING-Moduls

Dabei reserviert das Modul 4MByte Speicher für den Ringpuffer – das maximale, was der Kernel als zusammenhängenden physischen Speicher anbietet.

Anfangs wurden Versuchsreihen jeweils mit NPTL und LinuxThreads durchgeführt, die Unterschiede lagen jedoch unterhalb der Messgenauigkeit, was nicht besonders überraschend war – während die NPTL die Thread-Kommunikation beschleunigt, ist diese immer noch so langsam, dass ihre Vermeidung beim Programmentwurf eine hohe Priorität hatte und es nur wenige kritische Bereiche gibt. Die darauffolgenden Messungen wurden wegen der geringen Unterschiede daher nur noch mit LinuxThreads durchgeführt.

Es wurde jeweils gemessen, wie viele Pakete von IPAudit und flipacc in einem 100-Sekunden-Zeitraum ausgewertet werden können. Dabei wurde flipacc jeweils mit einem, zwei, vier und acht Analyse-Threads ausgeführt, um die Skalierbarkeit zu testen.

Außerdem wurden die Auswirkungen des PF_RING-Treibers untersucht, indem beide Programme einmal mit (bei flipacc direkt über das entsprechende Modul, und bei ipaudit über den Patch der libpcap) und einmal ohne den Kernel-Ring gemessen wurden.

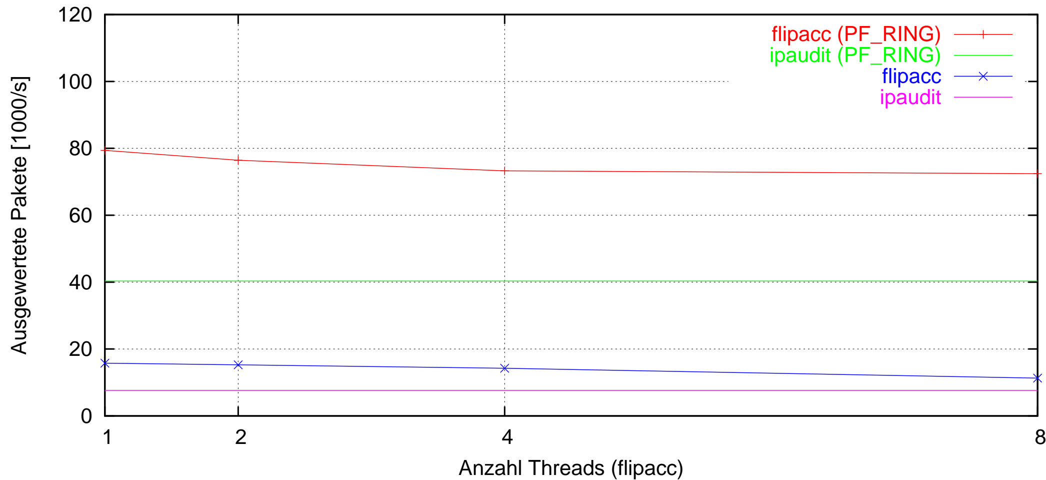

Abbildung 5.1: Tuning-Phase, 1 CPU (`nosmp`)

Für den ersten Durchlauf (Abb. 5.1) wurde eine der beiden CPUs über den Bootparameter `nosmp` deaktiviert, die zweite Messreihe (Abb. 5.2) fand mit zwei Prozessoren statt. Im ersten Diagramm sieht man, dass keine der Kombinationen alle Pakete verarbeiten konnte - hier ist die langsame CPU der limitierende Faktor.

Gleichzeitig sieht man aber, dass sowohl IPAudit als auch flipacc von dem direkten Empfang der Pakete über PF_RING profitieren, und die Anzahl der verarbeiteten Pakte um mehr als Faktor vier steigern. Dabei profitiert flipacc etwas mehr, da die Pakete dort direkt aus dem Ring gelesen werden und nicht mehr die libpcap dazwischen ist.

Die höhere Performance von flipacc gegenüber IPAudit erklärt sich unter anderem dadurch, dass IPAudit generische Analyse-Routinen hat, die zur Programmlaufzeit entscheiden, ob Informationen über Ports, MAC-Adressen und andere Faktoren zu den relevanten Daten zählen, und damit der Bearbeitungsaufwand pro Paket höher ist.

Werden beide Prozessoren eingesetzt (Abb. 5.2), steigt die Leistung aller Kombinationen deutlich an. Obwohl IPAudit nicht mehrprozessorfähig ist, kann der Kernel auf einer CPU die Netzwerkinterrupts bearbeiten, während die andere Pakete auswertet.

Hier ist flipacc mit PF_RING-Patch bereits in der Lage, praktisch den kompletten Traffic auszuwerten – ab zwei Threads sind die Paketverluste unter 1%, welches sich durch einen Ringüberlauf beim Programmstart (nach dem Start des Collectors, aber vor Beginn der Analyse) erklären lässt. Ohne den Ring-Patch profitiert flipacc dagegen leicht bei zwei Threads, vermutlich da bei mehr Threads die Caches der CPUs durch die zusätzliche Belastung

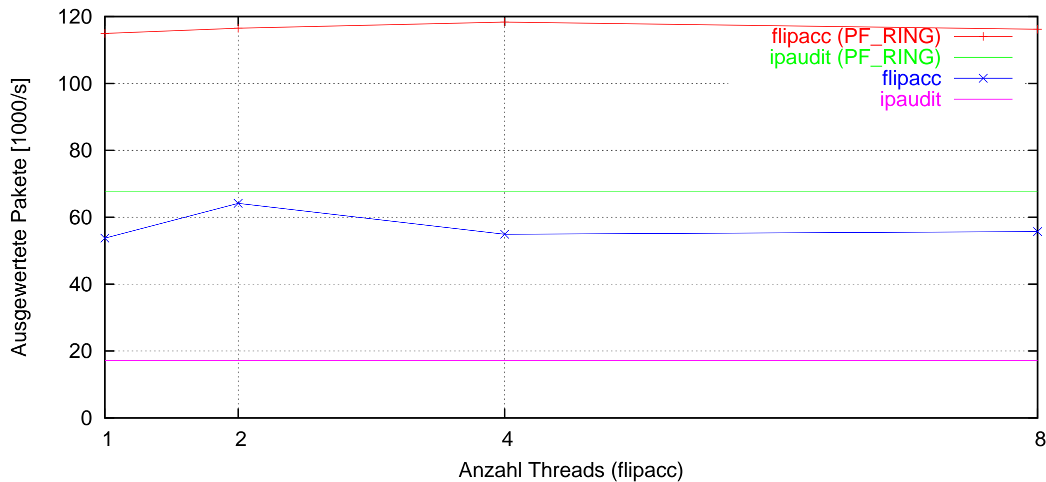

Abbildung 5.2: Tuning-Phase, 2 CPUs

mit der libpcap thrashed werden und mehr Zugriffe auf den langsamen RAM notwendig werden.

Die Relationen zwischen den Anwendungen bleiben erhalten, abgesehen davon dass flipacc mit PF_RING bereits den gesamten Traffic mitschneidet und CPU-Zyklen ungenutzt lässt, und damit nicht auf eine Vervierfachung kommt.

5.1.3 Fazit Tuningphase

Neben einer Optimierung der Modulparameter für PF_RING und der Feststellung, dass es für die Anwendung keine relevanten Unterschiede zwischen der eingesetzten Threading-Bibliothek gibt, haben die ersten Messreihen vor allem deutlich gemacht, dass die Kombination aus PF_RING und flipacc deutliche Performance-Reserven gegenüber anderen Lösungen bietet. Dagegen ist flipacc ohne den Ring-Treiber zwar generell einsetzbar, jedoch nicht für tatsächliches High-Performance-IP-Accounting geeignet.

5.2 Produktionstest

5.2.1 Versuchsaufbau

Für den Test in der Produktionsumgebung wurde ein Dual-Xeon (P4-Kern, 2800 MHz) mit 1GB Dual-Channel DDR-RAM und einer über PCI-X (100 MHz, 64 Bit) angebundenen 3Com 3C996 Glasfaser-Gigabit-Netzwerkkarte mit Tigon3-Chip von Broadcom verwendet.

Mit aktiviertem HyperThreading und dem SMT-Scheduler des 2.6.10er Linux-Kernels (und damit vier virtuellen CPUs) sollte dabei die maximale Rechenleistung zur Verfügung stehen.

Dieses Messsystem wurde über eine Glasfaserleitung an den Core-Router des Rechenzentrums angeschlossen, und der entsprechende Port wurde darauf konfiguriert, den gesamten Datenverkehr des Rechenzentrum-Uplinks zu spiegeln.

Da der Uplink eine höhere maximale Bandbreite hatte als das mit 1 GBit angebundene Messsystem, wurde die Netzwerkverbindung zwischen den beiden bei vielen Messreihen gesättigt. Das spielte für die Messungen selbst allerdings keine Rolle – lediglich beim Praxiseinsatz ist darauf zu achten, dass Pakete, die über 1 GBit/s hinausgehen, nicht mehr ausgewertet werden können.

Dadurch, dass die gerade tatsächlich von Rechnern im RZ übertragenen Daten empfangen und ausgewertet wurden, konnten jedoch keine Garantien für die konstante Beschaffenheit der Pakete über eine Messreihe hinweg gegeben werden.

Die Messreihen bestanden jeweils aus einem Lauf des ipaudit-Binaries, einem Lauf des gegen die PF_RING-nutzende libpcap-Version gelinkten ipaudit-Binaries und flipacc-Läufen mit 1, 2, 4, 8, 16 und 32 Analyse-Threads, die unmittelbar nacheinander durchgeführt wurden. Die flipacc-Tests wurden dabei einmal mit einer zentralen Hash-Tabelle und einmal mit thread-eigenen Tabellen durchgeführt. Angesichts der Ergebnisse aus dem ersten Versuch wurde auf flipacc ohne PF_RING dagegen vollständig verzichtet.

5.2.2 Messergebnisse

Insgesamt wurden über 50 Messreihen zu verschiedenen Uhrzeiten durchgeführt, so dass unterschiedliche Auslastungszustände des Netzes abgedeckt wurden. Es wurden damit Situationen mit 120.000 bis 500.000 Paketen pro Sekunde erfasst, da im Lauf der Messungen permanent Daten im RZ übertragen wurden. Während das Verhalten bei geringeren Datenraten eher unspektakulär ausfallen dürfte, wäre eine Messung unter DDoS-Bedingungen sehr interessant. Allerdings haben zur Messzeit keine Angriffe stattgefunden, und die künstliche Generierung solcher Paketmengen war ohne Spezialhardware nicht durchführbar.

Um von den veränderlichen Bedingungen auf dem Medium (Paketanzahl, Paketgrößen und Auslastung) möglichst gut abstrahieren zu können, wurde zur Auswertung eine Diagrammform gewählt, in der die Anzahl verarbeiteter Pakete über der Anzahl der empfangenen Pakete dargestellt wird.

Die Paketanzahl ist der beste Indikator für die tatsächlich anfallende Arbeit, da für jedes Paket (praktisch unabhängig von seiner Größe) ein konstanter Bearbeitungsaufwand entsteht. Die beiden Anwendungen wurden so erweitert, dass sie bei Beendigung die Anzahl der vom Kernel empfangenen und die der tatsächlich vom Programm ausgewerteten Pakete ausgeben.

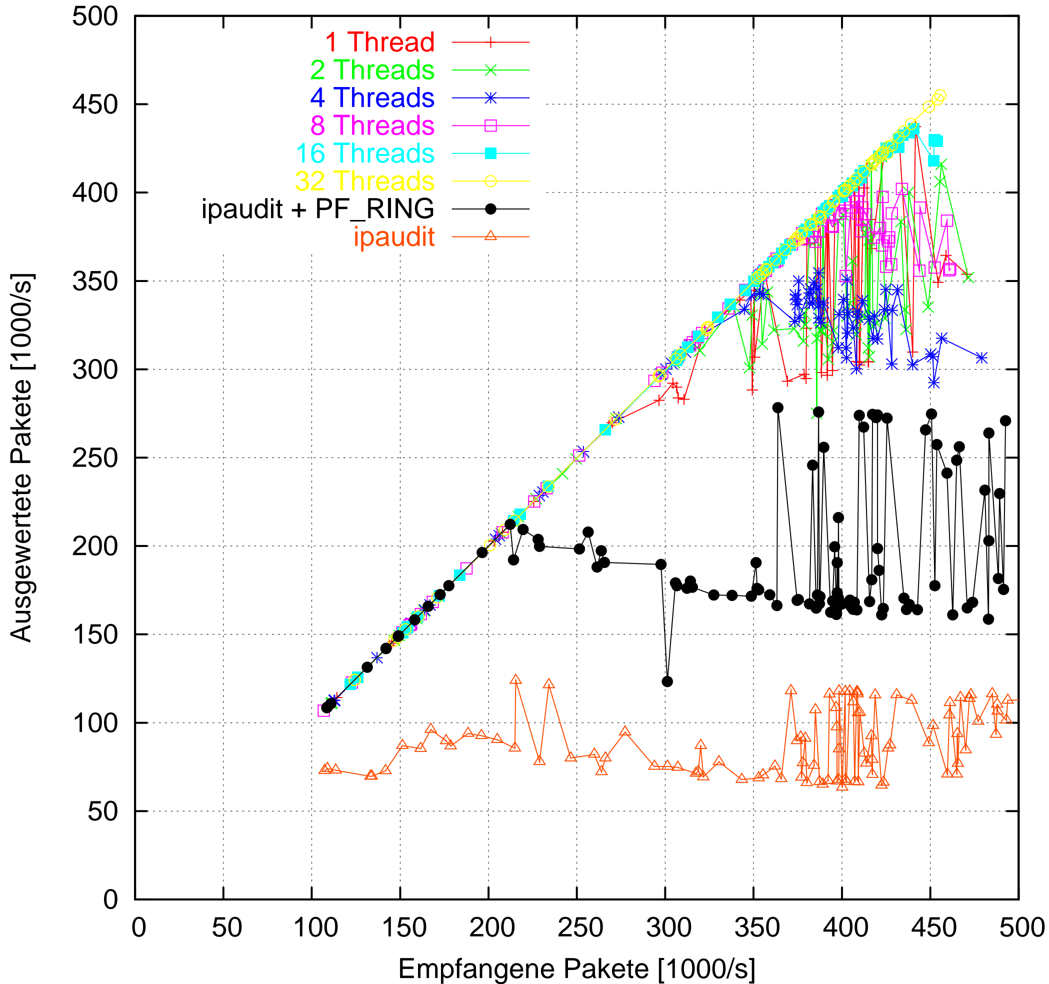

Abbildung 5.3: flipacc mit zentraler Hashtabelle / ipaudit

Dabei ergab sich für IPAudit ein relativ klares Bild: ohne die Beschleunigung durch PF_RING konnten im Durchschnitt höchstens 80.000 Pakete pro Sekunde ausgewertet werden. Mit Hilfe des Kernel-Treibers stieg der Wert auf über 170.000 Pakete (Abb. 5.3).

Mit flipacc dagegen konnten bei einer zentralen Hashtabelle bis zu 350Kp/s (Tausend Pakete pro Sekunde) ohne Probleme analysiert werden. Wurden jedoch mehr Pakete empfangen, so trat das Problem auf, dass die ausgewerteten

Paketmengen nicht mehr in direktem Zusammenhang mit den empfangenen Mengen standen. Da das Problem bei beiden Anwendungen und unabhängig vom PF_RING-Treiber bestand, liegt eine der Ursachen vermutlich im Kernel oder im Treiber der Netzwerkkarte.

So ist es wahrscheinlich, dass das System mit der Nachbehandlung der ca. 50.000-60.000 Interrupts pro Sekunde und dem darauf folgenden Re-Scheduling überfordert ist. Durch Beobachtungen während des Messablaufs wurde festgestellt, dass beide (realen) Prozessoren ca. 40% ihrer Zeit im Soft-IRQ-Kontext verbringen. Dieser Modus wird hauptsächlich dafür benötigt, Datenpakete von einem NAPI-Netzwerktreiber zu empfangen und in die Kernelstrukturen (bzw. im konkreten Fall nur in den Ringpuffer für die Auswertung) einzubinden.

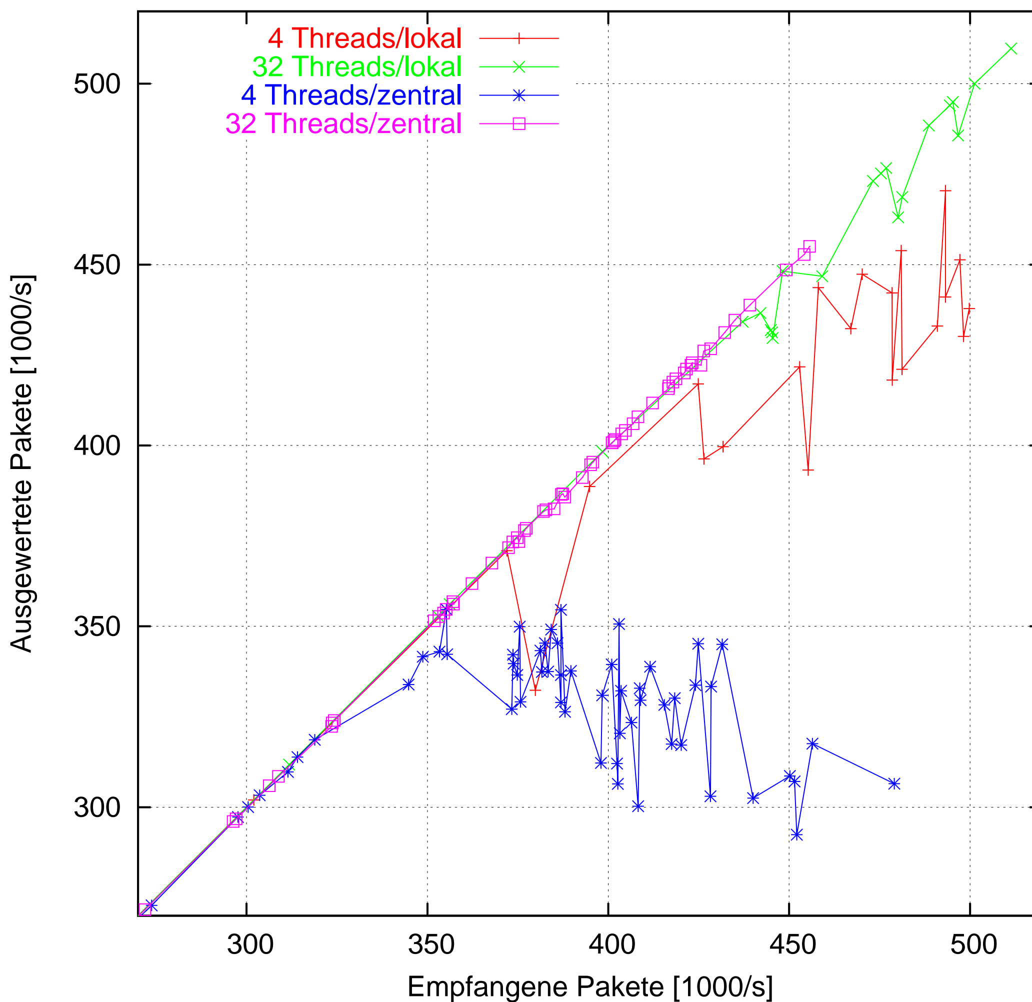

Abbildung 5.4: flipacc – zentrale vs. lokale Hashtabelle

Allerdings spielt bei diesen Paketdurchsätzen unter flipacc auch die Synchronisierung der Threads untereinander eine entscheidende Rolle. Wird eine zentrale Hashtabelle verwendet, kommen sich die einzelnen Auswerter häufiger in die Quere, und es gibt Paketverluste. Dieses Problem verringert sich zwar mit erhöhter Threadanzahl, verschwindet jedoch wahrscheinlich nicht ganz.

Auf der anderen Seite steht die Verwendung lokaler Hash-Tabellen: hier ist keine Synchronisierung mehr zwischen den auswertenden Threads notwendig, so dass kein hartes Limit mehr zu beobachten ist (in Abb. 5.4 zu sehen bei vier Threads mit zentraler Hashtabelle), sondern die Auswerterate weiter ansteigt. Durch die höhere notwendige Speicherbandbreite treten bei 32 Threads aber schon früher Verluste auf, als das bei einer von allen Threads geteilten Tabelle der Fall ist.

Das gesamte Problem ist jedoch nicht so gravierend wie es auf den ersten Blick erscheint, da im praktischen Einsatz schon viel früher die Höchstgrenze der Netzwerkkarte erreicht wird – wie in Diagramm 5.5 dargestellt, wurde bei den Messreihen mit über 270Kp/s das Bandbreitenlimit der Karte erreicht. Für das Diagramm wurden die Byte-Zähler der Messungen mit 16 Threads verwendet, da diese überall sehr geringe Verlustraten hatten.

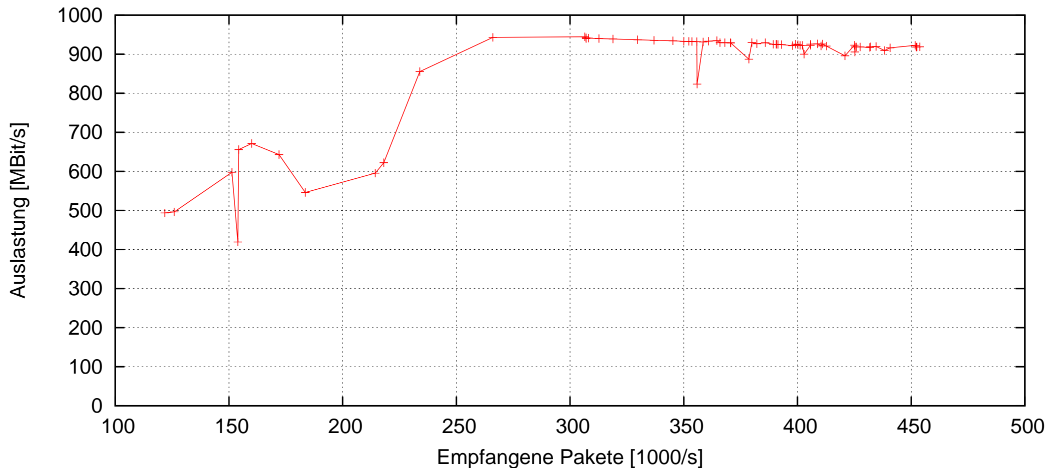

Abbildung 5.5: Gemessene Bandbreite

Dass dabei nur 920Mbit/s im Maximum gemessen wurden, erklärt sich dadurch, dass die Messsoftware weder die Präambel und die CRC-Checksumme des Pakets, noch die Intervalle zwischen Paketen signalisiert bekommt. Rechnet man diese mit ein, wird 1GBit/s erreicht.

Ist jedoch die Netzwerkkarte saturiert, wird die Datenquelle, also der Co-

re-Router, gezwungen Pakete zu verwerfen. In einem realen Einsatzszenario müsste das Accounting mit einer Gigabit-Ethernet-Karte deshalb an einer Stelle erfolgen, wo weniger als 1 GBit/s anfällt, damit keine Daten verloren gehen. Dort wären die Paketmengen bei normalem Betrieb auch entsprechend geringer.

5.2.3 Fazit

Im realen Einsatz skaliert flipacc ohne Probleme auf bis zu 350Kp/s, das ist die fünffache Auswerterate der bisher eingesetzten Software, bzw. etwa das Doppelte von dem, was mit IPAudit in Kombination mit dem PF_RING-Patch möglich ist.

Steigt die Paketrate noch weiter an, wird es immer schwieriger, alle Pakete auszuwerten – zum einen wird fast die Hälfte der Rechenzeit allein für die Behandlung der Netzwerkkarten-IRQs und die Einordnung der neuen Daten in den Ringpuffer verbraucht. Zum anderen wird der auf 4MByte (bzw. 65K Zellen) begrenzte Ringpuffer mehrere Male pro Sekunde komplett neu gefüllt, und muss von den Threads entsprechend schnell ausgelesen werden, um Paketverluste zu vermeiden.

Da bei steigenden Paketraten auch die Wahrscheinlichkeit zunimmt, dass zwei Threads den gleichen Datensatz bearbeiten wollen, profitiert hier die Lösung, bei der jeder Thread eine eigene Hashtable hat. Allerdings geht das auf Kosten von mehr Speichertransfers, so dass ein System mit einer langsameren RAM-Anbindung ein anderes Verhalten zeigen kann.

Wenn man pro CPU mehrere Threads verwendet, lassen sich dabei sogar bis zu 500.000 Pakete sekündlich auswerten, ohne dass merkliche Verluste auftreten. Höhere Paketraten ließen sich mit Bordmitteln nicht erzeugen, es ist jedoch davon auszugehen, dass die Anwendung noch etwas Spielraum bietet, bevor die Menge der verarbeiteten Pakete wieder abnimmt, weil die Kommunikation mit der Netzwerkkarte immer mehr Zeit benötigt und daher Pakete schon von der Hardware verworfen werden. Ein Zeichen für weitere Reserven auf Softwareebene ist auch, dass bei mehreren Messreihen über 480Kp/s alle Pakete mit nur einem Thread verarbeitet werden konnten, die zweite CPU also nur teilweise ausgelastet wurde.

Ein von Angreifern übernommener Nutzerrechner mit 100MBit-Netzwerkkarte ist in der Lage, bis zu 150Kp/s zu generieren – setzt man die Anwendung an einem Switch mit durchschnittlich 300Mbit/s Datendurchsatz und ca. 300 Bytes pro Paket (also 120Kp/s) ein, lassen sich neben dem regulären Verkehr auch mehrere gleichzeitig laufende Angriffe ohne Probleme erkennen und auswerten.

Kapitel 6

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde zunächst der grundlegende Aufbau eines Rechenzentrums vorgestellt und erläutert, wozu IP-Accounting beim RZ-Betrieb notwendig ist. Nach einem Aufzeigen der Notwendigkeit einer neuen Accounting-Anwendung wurden die zu lösenden Teilprobleme hergeleitet und formuliert. Neben der Aufnahme und Analyse von Paketen waren das die Zwischenspeicherung und die Ausgabe der Ergebnisse. Besondere Schwerpunkte bestanden in der Skalierbarkeit und Einsatzstabilität der Software.

Im zweiten Kapitel wurden einige Wege vorgestellt, auf denen Pakete unter Linux aufgenommen und analysiert werden können. Neben Schnittstellen zur eigenen Programmentwicklung (PCAP, netfilter, PF_RING) wurde auch auf fertige Lösungen (netfilter, NetFlow, IPAudit, nTop) eingegangen. Außerdem wurden auch andere Performance-Faktoren beleuchtet, die beim Capturing eine Rolle spielen – sei es das mit NAPI unter Linux implementierte Device Polling oder Packet Sampling. Auch generelle Systemeinflüsse wie der Typ des verwendeten Interrupt-Controllers und die auf einem System eingesetzte Thread-Bibliothek können sich auf die Arbeitsgeschwindigkeit einer Accounting-Anwendung auswirken und wurden daher betrachtet.

Im dritten Kapitel wurden Algorithmen erarbeitet, die zur Lösung der Teilprobleme dienen sollten. So wurde für die Paketaufnahme neben der üblichen libpcap auch ein PF_RING-Interface eingebaut, das mit sechsstelligen Paketmengen pro Sekunde nicht überfordert wird, und der dort gebotene Ringpuffer gleich in der Software verwendet, um Kopiervorgänge einzusparen. Die Auswertung der Paketdaten als zeitaufwändigste Teilaufgabe wurde auf mehrere Threads verteilt, um so von Multiprozessorsysemen profitieren zu können. Für die Analysezwischenstände fanden Hashtabellen ihre Verwendung als im gezielten Zugriff auf Datenelemente sehr schnelle Struktur. Schließlich wurden Methoden vorgestellt, die angesammelten Daten zu exportieren.

Das vierte Kapitel stellt schließlich einzelne Implementierungsdetails zum im Vorkapitel vorgestellten Programmentwurf vor. Es erläutert die Schnittstellen für die Programmierung eigener Module zum Datensammeln oder zum Export von Statistiken, zeigt die dafür notwendigen Datenstrukturen im Detail und geht auch auf einige Programminterna ein.

Im fünften Kapitel wurde das Programm einmal unter Laborbedingungen und einmal im praktischen Einsatz in einem Rechenzentrum evaluiert und mit der bis dahin verwendeten Analysesoftware in Relation gesetzt. Dabei wurde unter den selben Bedingungen ein um ein Vielfaches verbesserter Analysedurchsatz festgestellt, der bis über 500.000 Pakete pro Sekunde reicht.

Trotz der konsequenten Umsetzung des High-Performance-Ziels und der gezielten Parallelisierung der Programmausführung wurden aber auch die Grenzen des Gesamtsystems aus Hardware, Linux-Kernel und Accounting-Anwendung deutlich.

Ausblick

Obwohl die vorgestellte Anwendung die mit Hausmitteln – also x86-PC-Hardware – auswertbaren Paketmengen um einiges angehoben hat, ist das Ende noch lange nicht erreicht.

So sind im Handel bereits die ersten 10-Gigabit-Netzwerkkarten erschienen, und es ist abzusehen, dass sie in den nächsten Jahren immer mehr Verwendung finden. Die dort anfallenden Datenmengen reizen aber nicht nur den PCI-X-Bus aus (der auf aktuellen Serverboards mit 133 MHz und 64 Bit eine Burst-Datenrate von ca. 1GByte/s bietet) sondern stellen auch die CPUs mit der Abarbeitung der Pakete vor deutlich größere Herausforderungen. Während für die Verwendung als Server bisher Gigabit ausreichend ist, da so gut wie keine Peripherie-Geräte verfügbar sind, die diese Datenmengen liefern können, ist die Verwendung von 10GbE-Hardware für IP-Accounting in naher Zukunft durchaus vorstellbar.

Auch eine Entwicklung in die Gegenrichtung ist interessant – nämlich wie weit man die Hardwareanforderungen verringern kann, um den gesamten Datenverkehr einer Gigabit-Schnittstelle auswerten zu können. Die erste sich dabei stellende Frage ist natürlich, wie sich die Auswertung im Fall eines Denial-Of-Service verschlechtern würde, und welche Anteile der Paketmengen im Stresstest erfasst werden müssen. Im Zusammenhang damit sollte auch eine Evaluierung der auf dem Markt verfügbaren GbE-Netzwerkkarten und deren Treiber für Linux erfolgen, da diese bei steigenden Paketdurchsätzen einen immer höheren Einfluss auf die Gesamtperformance haben.

Es sind aber auch Optimierungen in der Software möglich. So ist der

von PF_RING gebotene Puffer durch eine Kernel-Beschränkung auf 4MByte begrenzt. Um diese Beschränkung zu umgehen und den Ring zu vergrößern, müssen einige Teile des Treibers umgebaut werden. Als Ergebnis ist dabei eine verbesserte Stabilität gegenüber Unregelmäßigkeiten und Peaks mit kleinen Paketen – insbesondere bei wenigen Analyse-Threads – zu erwarten.

Die Verwendung thread-lokaler Hashtabellen erfordert deutlich höhere Speicherbandbreiten. An dieser Stelle könnte man mit einem NUMA-System (Non Uniform Memory Architecture) wie einem Vier-Wege-Opteron, wo jede CPU lokale Speicher hat, den Engpass auflösen. Entwickelt man den Gedanken weiter, könnte auch der Ringtreiber so geändert werden, dass jeder Thread einen eigenen Ringpuffer erhält, und die Verteilung der Daten dynamisch zur Laufzeit geregelt wird.

Listings

4.1 `struct capturer` - Beschreibung eines Packet Capturer . . . 26
4.2 Beispiel der Initialisierung eines Capturing-Moduls 27
4.3 `struct cpt_ring_slot` - Aufbau einer Ringzelle 27
4.4 `struct pcap_pkthdr` - Metadaten für ein empfangenes Paket 28
4.5 `struct hashtable` - Metadaten einer Hash-Tabelle 29
4.6 `struct anl_counter` - Aufbau Paketzähler 30
4.7 `struct hash_ip4_key` - IPv4-Protokolldaten 30
4.8 `struct hash_ip6_key` - IPv6-Protokolldaten 31
4.9 `struct anl_control` - Kontrollstruktur für Analyse-Threads 32
5.1 Initialisierung des PF_RING-Moduls 35

Abbildungsverzeichnis

3.1 Datenfluss beim Accounting 16
3.2 Ringpuffer mit zwei Analyse-Threads 19
3.3 Hash-Tabelle für Paketzähler (64K Einträge) 22

5.1 Tuning-Phase, 1 CPU (`nosmp`) 36
5.2 Tuning-Phase, 2 CPUs 37
5.3 flipacc mit zentraler Hashtabelle / ipaudit 39
5.4 flipacc – zentrale vs. lokale Hashtabelle 40
5.5 Gemessene Bandbreite 41

Selbständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Studienarbeit „High-Performance IP-Accounting unter Linux" selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, sowie alle Zitate entsprechend kenntlich gemacht habe.

Magdeburg, 18.01.2005
Georg Lukas

Literaturverzeichnis

[1] Packet Capture library (PCAP)
http://www.tcpdump.org/

[2] The netfilter/iptables project
http://www.iptables.org/

[3] IPAC-NG - IP accounting next generation
http://ipac-ng.sourceforge.net/

[4] „Improving Passive Packet Capture:Beyond Device Polling", Luca Deri
http://luca.ntop.org/Ring.pdf

[5] ISO/IEC-Standard 9945-1:1996 (auch ANSI/IEEE POSIX 1003.1-1995, insbesondere 1003.1c), Dokumentreferenz-Nr. im IEEE publications catalogue: SH 94352-NYF

[6] „Algorithm Alley", Bob Jenkins, Dr. Dobbs Journal, September 1997.
Quellcode: http://burtleburtle.net/bob/hash/

[7] „Denial of Service via Algorithmic Complexity Attacks", Scott A. Crosby und Dan S. Wallach. In Proceedings of the 12th USENIX Security Symposium, S. 29–44, August 2003.
auch http://www.cs.rice.edu/~scrosby/hash/

[8] „Algorithmic Complexity Attacks and the Linux Networking Code", Florian Weimer
http://www.enyo.de/fw/security/notes/linux-dst-cache-dos.html