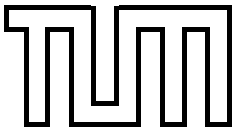
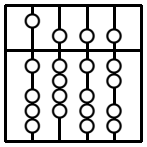


TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK



Analyse und Umsetzung einer Filter-basierten Paketverarbeitungsmaschine für IP-Netzwerke

Systementwicklungsprojekt

Tobias Sandhaas

Aufgabensteller:	Prof. Dr. Dr. Uwe Baumgarten
Betreuer:	Prof. Dr. Dr. Uwe Baumgarten
Abgabedatum:	26. September 2002

Inhaltsverzeichnis

1 Einleitung

Das Internet erlebte in den letzten Jahren einen stetigen Boom und fand zu einer neuen Akzeptanz sowohl bei Privatanwender als auch in Wirtschaftsunternehmen. Dieser starke Zuwachs wurde von viele Internet Service Provider (ISP), die als Mittler zwischen Endkunden und dem Backbone Carriern des Internets fungieren, durch aggressive Marktpolitik durch Preisdumping gefördert. Vielerorts bestand die Möglichkeit den Zugang zum Internet über Transfervolumen unabhängige Abrechnungsmodelle (Flatrate) abzurechnen. Diese Modelle sind auf Dauer nicht zu finanzieren, weshalb der Bedarf verbrauchsorientierter Abrechnung[1] zunehmend in den Vordergrund rückt. Um dies zu ermöglichen benötigt man eine Softwarelösung, die effizient alle IP-Pakete den betreffenden Kunden zuweist, und letztendlich eine korrekte Rechnungsstellung ermöglicht.

Aufgrund sehr variabler Netzstrukturen innerhalb eines ISPs, vor allem jene die multi-homed [2] an das Internet angebunden sind, muss ein verteiltes Datenerfassungssystem entwickelt werden, welches an mehreren Messpunkten die Daten sammeln und korrekt aufsummieren kann. Zudem muss das System sehr effizient arbeiten damit auch relativ große Datenströme ohne Verluste erfasst werden können. Es sollte äußerst fehlertolerant und unter realen Bedingungen längere Zeit im Probebetrieb getestet worden sein, da Ausfälle oder Fehler in der Datenerfassung vermieden werden müssen.

Ziel dieses Systementwicklungsprojekts ist eine funktionsfähige "Filter-basierte Paketverarbeitungsmaschine für IP-Netzwerke"[3] zu erstellen, welche den obigen Bedingungen genügt. Hierzu ist zunächst erforderlich wichtige Faktoren, die die Effizient dieses Systems beeinflussen zu lokalisieren und deren Grenzwerte, die einen einwandfreien Betrieb gewährleisten, zu ermitteln.

Das Projekt wird bei dem ISP "netplace Telematic GmbH" durchgeführt, die uns den Zugriff auf ihr Netz als auch ihr Equipment zur Verfügung stellt. Die funktionsfähige Software soll später mitunter auch dort im Einsatz sein.

[1] IP Accounting
[2] mehrere unabhängige Zugänge an das Internet besitzen
[3] engl.: filter-based packet capturing engine on IP-networks

2 Analyse wichtiger Faktoren

2.1 Die Testumgebung

Um einflussnehmende Faktoren bei der Datenerfassung bewerten zu können, wurde eine möglichst einheitliche Testumgebung aufgebaut. Als zentraler Austauschpunkt der IP-Pakete dient in unserem Netz ein “Catalyst 2950T-24“ Switch von Cisco mit neun 100MBit und einem 1GBit Port. Über die 100MBit Ports werden IP-Pakete in das Netz injiziert und an einen Rechner der ebenfalls an einem dieser 100MBit Ports hängt versendet. Der 1GBit Port ist als Monitoring-Port konfiguriert und empfängt daher Duplikate sämtlicher Pakete die in dem Netz fließen. An diesem Port wird über eine 3com 3c996B-T Netzwerkkarte ein Dual-Prozessor Intel-PC mit einem GHz Prozessortakt angeschlossen, der einem Sniffer ähnlich alle IP-Pakete im Netz abhorcht. In Abb. 2.1 ist der strukturelle Aufbau der Testumgebung aufgezeichnet.

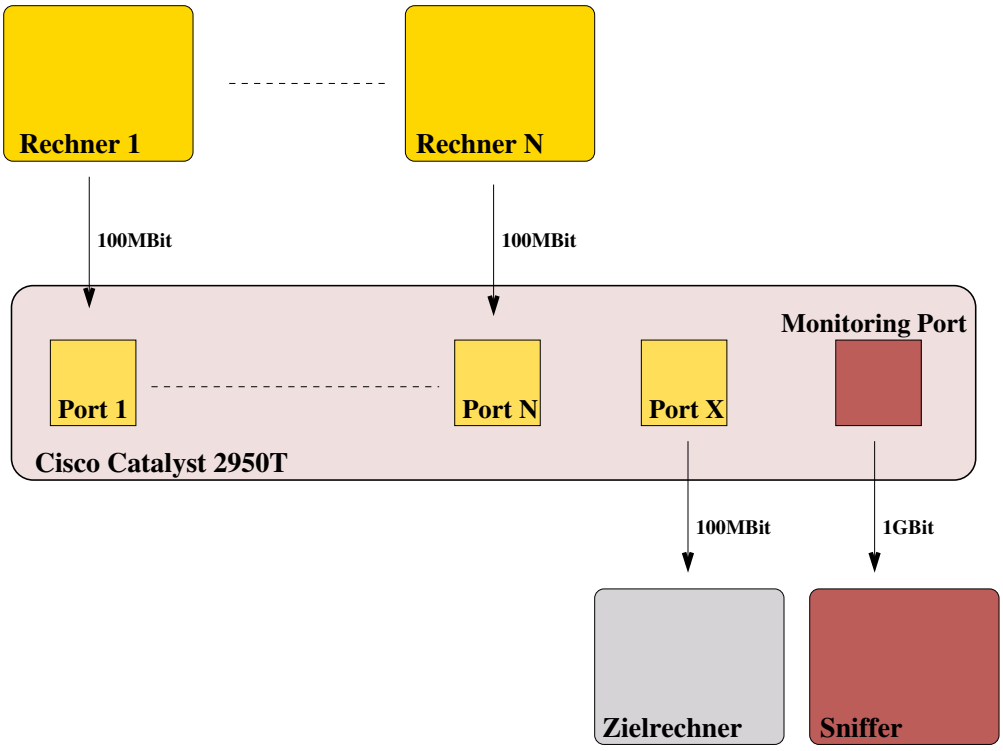


Abbildung 2.1: Struktureller Aufbau der Testumgebung

Grundsätzlich verschicken die N-Quellrechner 576 Bytes[1] große UDP Pakete an den Port 10000 des Zielrechner. Diese Pakete werden nun auf beiden Seiten gezählt, also sowohl die Summe der verschickten Pakete als auch die Anzahl der empfangenen Pakete am Sniffer, der eben die Duplikate erhält. Man verwendet den Port 10000 um die UDP Pakete von anderen im Netz auftretenden UDP-Paketen unterscheiden zu können und das Ergebnis nicht zu verfälschen.

Ausschlagender Faktor unserer Versuche ist also die Differenz dieser beiden Summen, sprich die Anzahl der verlorenen Pakete. Da lediglich der Fehler am Sniffer selber bewertet werden soll, müssen all die Faktoren, die vor dem Sniffer zu Verlusten führen so gut es geht ausgeschlossen werden:

- Verluste beim IP-Paket Sender
  Da jeder Sender von IP-Paketen an einen dedizierten 100MBit Port hängt und diese Rechner sonst kaum zusätzliche Pakete erzeugen, steht theoretisch auch diese Bandbreite zur Verfügung. Interne Versuche zeigten uns, dass unsere Hardware praktisch bis zu 96,2 MBit/s an UDP Traffic verschicken kann. Sicherheitshalber werden wir aber jeden einzelnen Rechner nicht mit mehr als 80 MBit/s im Maximum ausreizen.
  Auf jedem dieser Rechner wird ein eigens entwickelter TCP-Dämon gestartet, welcher auf Kommandos lauscht und gegebenfalls mit dem Injizieren von UDP-Paketen in die Testumgebung startet. Wir übergeben diesem Dämon hierbei die Paketrate, sprich Pakete in der Sekunde, als auch die Zeitdauer für diese Aktion. Um einen möglichst gleichmäßigen Ausstoß (ohne Spitzen) von Paketen zu erzielen, verwendet das Programm einen Signalhandlers vom Typ `SIGALRM`, der bei Aufruf eine zuvor festgelegte Anzahl von Paketen über einen Raw-Socket verschickt. Da in der Testumgebung lediglich Linux eingesetzt wird und Linux bekanntlich kein Realtime OS ist kann dieser Signalhandler aufgrund von Systembeschränkungen[2] nur minimal alle 10ms gerufen werden. Dadurch ergaben sich bei zu großen Paketen inhomogene Senderaten, da die zu verschickende Datenmenge durch wenige Pakete stets am Anfang eines Aufrufs verschickt werden konnten. Bei zu kleinen Paketgrößen hingegen konnte das Betriebssystem die benötige Anzahl an Paketen nicht innerhalb eines Aufrufs versenden. In beiden Fällen kam es zu Paketverlusten am Sender. Für die Testreihen legt man daher die Paketgrößen auf 576 Bytes fest, da ein Großteil der Pakete im Internet ohnehin diese Größe einnehmen und Probeversuche zeigten, dass obige Probleme hierbei zu keinem Paketverlust führt. Aufgrund dessen werden nur maximal 20000 Pakete in der Sekunde pro Sender zugelassen.

- Verluste am Switch
  Da wir mit dem Cisco Catalyst 2950T ein ausgereiftes Produkt einsetzen und interne Versuche zuvor bestätigt haben, dass dieser Switch ohne Probleme mehreren 100 MBit Strömen ohne Verluste verarbeiten kann, kann diese potentielle Fehlerquelle nahezu ausgeschlossen werden. Leider

[1] Pakete mit maximal 576 Bytes Größe werden im Internet laut []-Norm nicht fragmentiert
[2] System timer resolution: kleinste ununterbrechbare Aktionseinheit(jiffy)

ist die Datenerfassung am Switch, welche mittels SNMP abgefragt werden kann, nicht immer 100% korrekt (tatsächlich werden Pakete teilweise nicht gezählt) und stellt daher keinen Referenzwert für die Summe an verschickten Paketen dar.

Der Sniffer wurde in Hinblick auf Genauigkeit der Datenerfassung programmiert. Als Basis dieses Systems dient uns die libpcap[3], welche einen vereinfachten Zugriff auf netzwerkrelevante Funktionen des Betriebssystem ermöglicht. Für die Kommunikation mit den Routern wird die SNMP Bibliothek ucd-snmp[4] verwendet.

2.2 Einflussnehmende Größen

Da in diesem Projekt eine Vielzahl von Hard- und Softwarekomponenten eingesetzt werden, ist eine Analyse aller Größen, die die Versuchsreihen beeinflussen unmöglich. Im Folgenden werden einige wichtige Komponenten herausgegriffen und analysiert.

2.2.1 Betriebssysteme

Das Betriebssystem ist die wichtigste Komponente und beeinflusst die Messergebnisse daher am stärksten. Wir werden daher die Ergebnisse von FreeBSD 4.5 und Redhat Linux 7.2 mit dem Kernel 2.4.7 gegenüberstellen. Als Grundlage für eine Analyse bzw. Optimierung von Kernelparameter muss man den Weg eines IP-Paket durch den betreffenden Kernel bis hin zum Applikations-Socket kennen. Da wir unveränderte IP-Pakete verarbeiten wollen verwendet man in diesen Fällen den `PF_Paket` Socket. Bei Verwendung dieses Sockets werden keine weiteren Protokoll-Routinen[5] Funktionen aufgerufen, die das Paket ansonsten weiter untersuchen und verändern würden.
Ein eingehendes Ethernet-Paket wird zunächst von der Hardware der Netzwerkkarte empfangen und in dessen Speicher zwischengespeichert. Sollte das Paket vollständig (oder unfragmentiert) vorliegen wird ein interrupt request(IRQ) an den Netzwerkkartentreiber `ei_interrupt()` weitergeleitet.
Dieser veranlasst daraufhin, dass weitere Interrupts im Betriebssystem ignoriert werden, solange das Paket nicht aus der Hardware in die `input packetqueue` kopiert wurde. Das Paket wird also zunächst, meistens mittels DMA, in ein im Kernelspeicher angelegten `struct sk_buff` kopiert. Dieser Container repräsentiert die Ansicht des IP-Pakets auf Kernelebene und wird bis das Paket an die Benutzerapplikation übergeben wird nicht mehr freigegeben. Daraufhin wird die generische Empfangsroutine[6] `netif_rx()` aufgerufen, welche die Aufgabe hat

[3] TCPDUMP public repository - http://www.tcpdump.org/
[4] NET-SNMP Project - http://net-snmp.sourceforge.net/
[5] protocol handler
[6] generic network reception handler

das Paket aus dem Netzwerktreiber in die Warteschlange (`input packetqueue`) einzureihen und einen `softIRQ` zu auszulösen. Die `input packetqueue` ist der zentrale Sammelpunkt der Pakete aus allen Netzwerktreibern und hat ein festgelegtes Fassungsvermögen von `netdev_max_backlog` Paketen.
`softIRQs` werden seit Kernel 2.4 eingesetzt und werden immer dann verwendet, wenn man den Einsatz von Interrupts, welche ununterbrechbar sind, vermeiden will, aber zuvor registrierte Funktionen (handler) möglichst bald ausgeführt werden sollen. Das Betriebssystem überprüft mittels `do_softirq()` ob Anfragen vorliegen und führt dann zuvor registrierte Routinen aus. Die betreffende Routine für Netzwerk Pakete ist `net_rx_action()`. Um zu gewährleisten, dass diese Anfragen ohne lange Wartezeiten bedient werden, wird die `do_softirq()` Funktion nach jedem "Hardware Interrupt" (kernel/irq.c), Systemcall(kernel/entry.S) und Scheduling eines neuen Prozesses (kernel/sched.c) aufgerufen. `net_rx_action()` nimmt nun so viele Pakete aus der `input packetqueue` und gibt sie an die Protokoll-Routinen weiter bis die Warteschlange leer oder ein jiffy (10ms) vergangen ist. Sollte die Warteschlange dann noch nicht leer sein wird der `softIRQ` erneut aktiviert. Da wir `PF_Paket` Sockets einsetzen ist die Protokoll-Routine `packet_rcv()` für die Weitergabe der Pakete aus der `input packetqueue` an die `receive queue` zuständig. Die `receive queue` ist ein Teil des Applikations-Sockets auf Kernelebene und wird durch `rmem_max` in der Größe beschränkt.
Die Pakete können nun von der Applikation mittels der `recv*()` Funktionen übernommen werden. Abb. 2.2 veranschaulicht den Paketfluss innerhalb des Kernels.

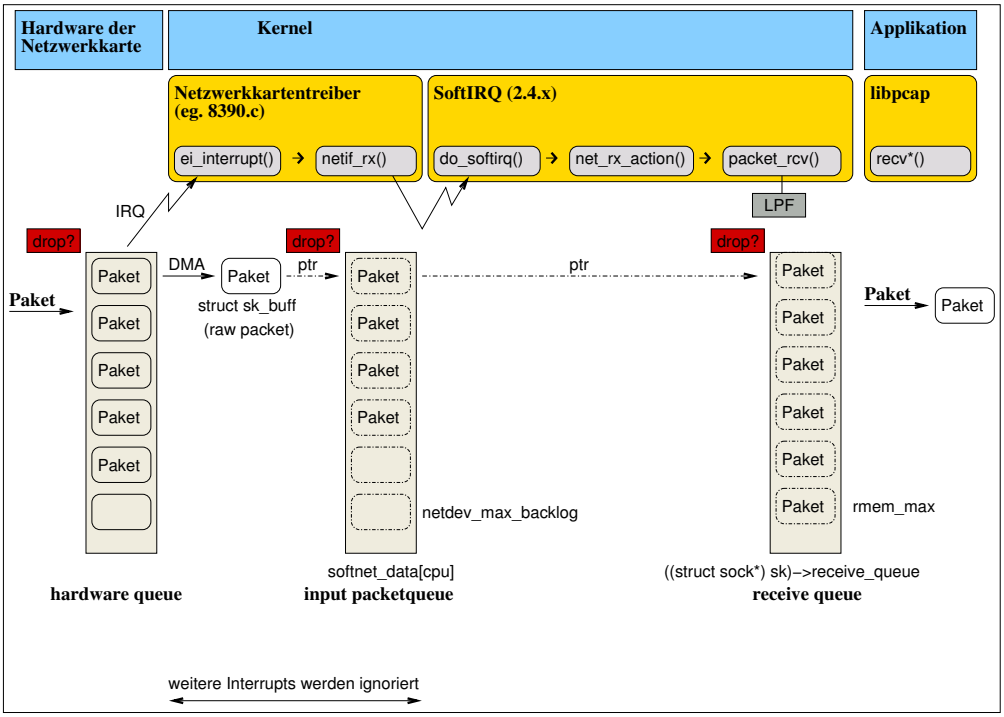


Abbildung 2.2: Paketfluss innerhalb des Linux Kernel

Prinzipiell können an drei Stellen Paketverluste auftreten:

- Verluste an der Netzwerkkarte durch Überlauf der `hardware queue`
  Dieser Bereich ist stark von der Hardware abhängig. Daher fällt eine Beurteilung schwer. Da der Versuch aber nicht fragmentierte UDP-Pakete verwenden, und diese daher nicht bestätigt werden müssen (`ACK`), dürfte der häufig auftretende interrupt request an die `ei_interrupt()` Funktion regelmäßig zu einer vollständigen Leerung der Warteschlange führen[7].

- Verluste durch Überlauf der `input packetqueue`
  Solange die `input packetqueue` mindestens die Größe der `hardware queue` hat, können Verluste an dieser Stelle ausgeschlossen werden. Denn sobald neue Pakete an der Netzwerkkarte eintreffen und über `netif_rx()` der `input packetqueue` hinzugefügt werden können, wird ein IRQ erzeugt, welcher zuvor über die `softIRQs` eine Leerung der `input packetqueue` bewirkt. Eine Testreihe bei dem ein modifizierter Kernel eingesetzt wurde, der uns die Fülle der `input packetqueue` protokolliert, bestätigte diese Schlussfolgerung.
  Zu erwähnen ist hierbei, dass durch `softIRQs` SMP-Systeme profitieren, denn die `input packetqueue` wird an jeden Prozessor gebunden und ermöglicht somit einen nebenläufigen Paketempfang.

- Verluste durch Überlauf der `receive queue`
  Da sämtliche Schritte des Pakets von der Netzwerkkarte bis zur `receive queue` innerhalb des Kernels stattfinden und die Durchlaufzeiten dementsprechend relativ gering sind, besteht an der Schnittstelle zur Benutzerapplikation, dessen Laufzeiten unter Linux nicht garantiert werden, potentiell Gefahr eines Überlaufs mit Paketverlusten. Um diese Auswirkungen zu reduzieren sollte man kritischen Anwendungen stark Priorisieren damit möglichst viel Rechenzeit trotz des Linux üblichen "fair-queueing" zur Verfügung steht. Dies erreicht man beispielsweise mit `sched_setscheduler()` und `nice`. Zudem besteht die Möglichkeit den Kernel zusätzlich durch Modifikationen am Quellcode zu optimieren. Hierbei bietet sich der `low-latency patch`[8] und `preemption patch`[9] an. Der `low latency patch` fördert den Kontextwechsel indem an mehreren Stellen des Kernels ein `schedule()` getätigt wird. Der `preemption patch` ermöglicht den Kernel einen aktiven Prozess zu Gunsten eines höher priorisierten zu unterbrechen. Da Stabilität für kritische Programme wichtig ist und beide Patches noch nicht vollends ausgereift sind und deshalb wohl den Standard 2.4.x Kernel noch nicht angehören, verzichten wir zunächst auf diese Erweiterungen. Dennoch haben wir Versuche auch mit diesen Patches durchgeführt um einen Eindruck der Performancesteigerung vorab zu gewinnen. Auf andere Ansätze, die Linux realtime fähiger machen (siehe [Dan01]) wurde ebenfalls verzichtet, da oft gravierende Änderungen an der Programm-API einher gingen.

[7] vorausgesetzt das System geht dem IRQ nach
[8] http://www.zip.com.au/~akpm/linux/schedlat.html#downloads
[9] http://www.tech9.net/rml/linux/

Eine andere Möglichkeit die Überläufe zu reduzieren besteht darin die `receive queue` (`rmem_max`) zu vergrößern. Dies kann problemlos mittels `setsockopt( *, *, SO_RCVBUF, *, *)` auf Applikationsebene eingestellt werden und ist sinnvoll um gravierende Transferschwankungen und Spitzen abzufangen.
Zusätzlich besteht die Möglichkeit den so genannten Linux Paket Filter (`LPF`) einzusetzen, der auf Kernelebene noch bevor das Paket die `receive queue` erreicht Pakete filtern kann. In unserem Szenario haben wir aber ein nahezu abgekapseltes System, sodass die wenigen zusätzlichen Pakete kaum ins Gewicht fallen dürften und wir daher nicht filtern.

2.2.2 Software

Aufgrund des vorausgehenden Kapitels sollte die Software möglichst schnell und effizient die Datenmenge in Form der Pakete verarbeiten können. Dies setzt voraus das beispielsweise möglichst keine blockende Syscalls, die man vor allem auf Netzwerkebene findet, eingesetzt werden. Die libpcap erlaubt das Abschneiden von Paketen der `receive queue` noch bevor sie an die Applikation weitergegeben werden und ermöglicht dadurch ein effizientere Verarbeitung. Da unsere System nur an dem UDP-Header interessiert ist, schneiden wir den Datenteil ab dem Datenteil, sprich dem 57 Byte (siehe [Dyk01]), ab.

2.3 Messergebnisse

Primäres Ziel der Versuchsreihen ist die prozentualen Verluste der UDP-Pakete in Abhängigkeit der Paketrate festzustellen. Hierbei wurde jeder Messpunkt zweimal geprüft und 5 Minuten pro Durchlauf veranschlagt. Als äußerstes Limit der systemweiten Paketrate wurde 90.000 Pakete pro Sekunde mit einer Größe von 576 Bytes vorgegeben. Dies entspricht einen Gesamtfluss von 396 MBit pro Sekunde.
In allen Testläufen wurden die betreffenden Anwendungen hoch priorisiert mittels `sched_setscheduler(0,SCHED_RR,MAX_PRIORITY)` und `nice --20` um ihr möglichst viel Rechenzeit zur Verfügung zu stellen.
Wie bereits in Kapitel 2.2.1 erwähnt ergaben Versuche mit einem modifizierten Linuxkernel, der die Fülle der `input packetqueue` protokolliert, dass eine Feinabstimmung hier nicht erforderlich ist. Daher wurde diesen Faktor bei den weiteren Versuchen nicht beachtet.
Besonderen Augenmerk wurde letztendlich auf die Größe der `receive queue` gelegt.

Es wurden drei verschiedene Kernel miteinander verglichen:

1. Linux Standardkernel 2.4.7 ohne Erweiterungen wie er von kernel.org erhältlich ist

2. Erweiterter Linux Standardkernel 2.4.7 um den low-latency und den preemption patch
3. FreeBSD 4.5 Standardkernel ohne Erweiterungen

Die erste Testreihe vergleicht diese Varianten mit einer `receive queue` von 64KByte miteinander. Der Linux Standardkernel hat hierbei auffällig einen Einbruch ab ca 70000 Pakete pro Sekunde, wobei FreeBSD einen gleichmäßigeren, aber auch höheren, Verlauf aufweist. Der modifizierte Linuxkernel, der der Benutzerapplikation besonders oft Rechenzeit zuweist, liefert mit Abstand die besten Ergebnisse mit durchwegs rund 2% Paketverlust. Dies lässt vermuten, dass die receive queue bei hohen Paketraten überläuft, wenn die Benutzerapplikation nicht oft genug die Chance erhält die `receive queue` zu leeren. Linux hat Abb. 2.3 veranschaulicht diese Ergebnisse.

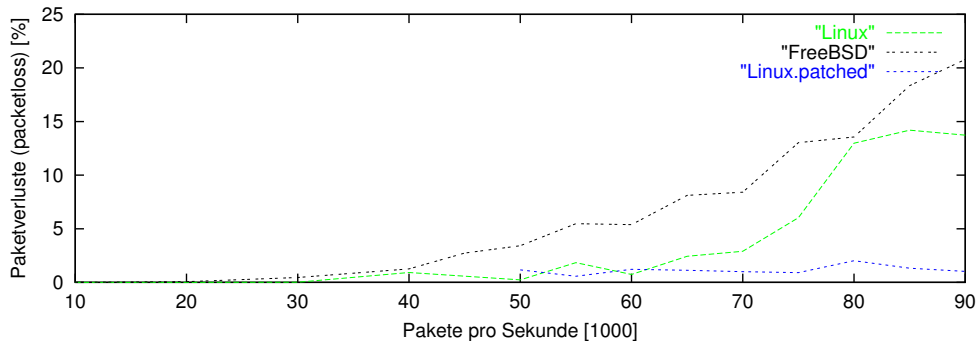


Abbildung 2.3: Diverse Kernel mit einer receive queue von 64 KByte

Daher wird in der zweiten Testreihe (Abb. 2.4) die Auswirkung verschieden großer `receive queues` unter dem Linux Standardkernel untersucht. Man erkennt deutlich, dass eine Vergrößerung auf 128KByte zu wesentlich besseren Ergebnissen mit nur 2% Verlusten führen. Die Ergebnisse bei 4MB unterscheiden sich nicht merklich zu den der 128KByte, da bei den verwendeten Paketraten und dem Linux Standardkernel die Rechenzeit der Benutzerapplikation anscheinend allemal ausreicht um selbst bei 128KByte die `receive queue` zu leeren.

Bleibt zu untersuchen, wie der Standardkernel mit großer `receive queue` im Vergleich zum erweiterten Linuxkernel abschneidet. Abb. 2.5 fasst das Ergebnis zusammen. Wie zu erwarten liefern beide Varianten sehr gute Ergebnisse mit Verlusten unter 2%. Keine der beiden Varianten schneidet daher in unserem Szenario besser ab. Die auftretende Verluste werden wahrscheinlich durch Sendespitzen am Paket-Sender hervorgerufen. Da wir bei 90.000 Pakete pro Sekunde in einem Zeitraum von 5 Minuten 27 Millionen Pakete verschicken, stellt ein Verlust von 1% gerade mal 270.000 verlorene Pakete dar. Daher sind weitere Fehlerquellen bei dieser geringen Größenordnung nicht auszuschließen.

Aufgrund dieser Ergebnisse und dem bereits bestehenden Know-how unter Linux haben wir uns für Linux und gegen FreeBSD als primäre Plattform der

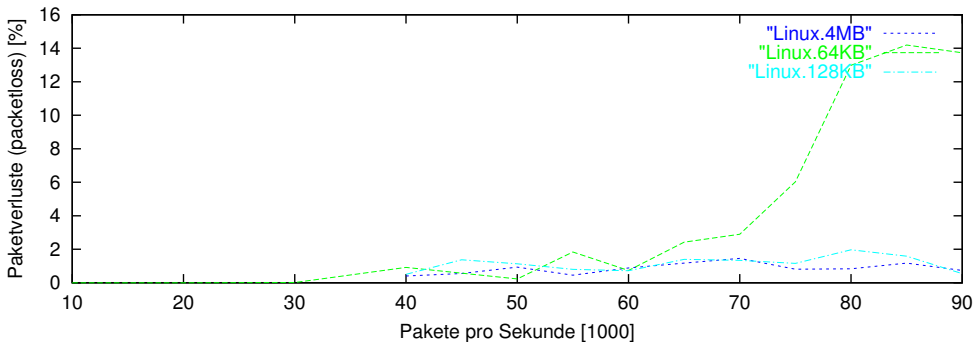


Abbildung 2.4: Standardkernel mit unterschiedlicher receive queues

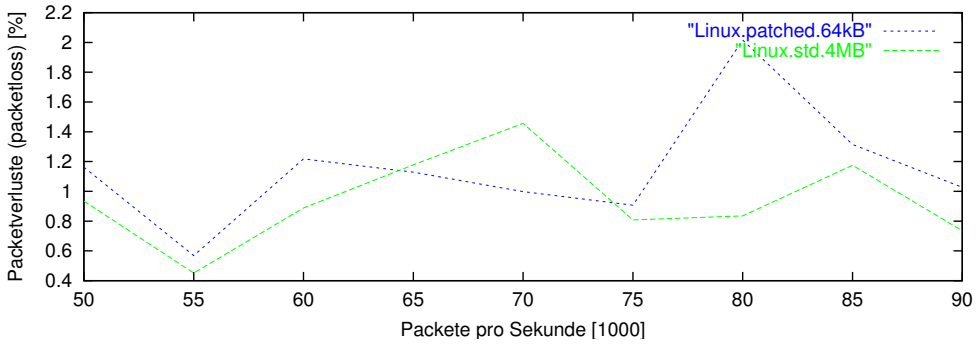


Abbildung 2.5: Erweiterte Linuxvariante mit 64KByte und Standardkernel mit 4MB receive queue

zu entwickelten Paketverarbeitungsmaschine entschieden. Wir werden ebenfalls wie auch schon in den Testreihen zugrunde liegenden Einstellungen um die Applikationen zu priorisieren vornehmen. Die Ergebnisse bestätigen unser Vorhaben einen Standardkernel einzusetzen, der nicht um besondere Fähigkeiten (Realtime Linux) erweitert werden muss. Bei Dimensionen bis zu 90.000 Pakete pro Sekunde und einer gemittelten Paketgröße von 576 Bytes reicht uns eine Vergrößerung der `receive queue` vollends aus.

3 Design

3.1 Grundsätzlicher Aufbau mit Funktionsbeschreibung

Das Hauptanliegen unseres Systems zur IP-Paketverarbeitung ist die verlustfreie Datenerfassung. Wie wir in der vorhergehenden Analyse in Kapitel 2 erfahren haben sollte die Applikation möglichst immer bereit sein Daten aus der `receive queue` zu nehmen. Da wir aber noch mit anderen Applikationen wie beispielsweise Datenbanken kommunizieren müssen und dieser Vorgang in der Regel asynchron ist, haben wir das System in drei Teilkomponenten getrennt:

- packet capture engine (`PCE`)
- data access engine (`DAE`)
- `data collector`

Primäre Aufgabe der `PCE` ist die Erfassung aller Datenströme im Netz und das Ablegen dieser in zwei im shared memory Bereich (`SHM`) liegender Container (`IP pool`) damit die DAE darauf zugreifen kann. Hierbei werden zugehörigen Pakete zusammengefasst um die anfallende Datenmenge zu reduzieren. Die Datenströme haben generell zwei Richtungen: Ausgehende Pakete, die vom Kundenrechner weg fließen werden nach der Source-IP eingeordnet und eingehende Pakete -zum Kundenrechner hin -nach der Destination-IP. Mittels Semaphoren (siehe Kapitel 3.3) wird sichergestellt, dass die PCE und DAE nicht zeitgleich auf einen dieser Container zugreift. Wenn der Bedarf besteht kann die PCE zudem IP-Pakete filtern, beispielsweise wenn nur Teilnetze erfasst werden sollen. Die PCE soll keine weiteren Tätigkeiten nachgehen um mit Hilfe der Priorisierungen so gut wie möglich die `receive queue` leeren zu können.

Die `DAE` nimmt die Pakete aus dem `SHM` entgegen und summiert diese nach Kundenzugehörigkeit so lange in einem speziellen Container auf bis der data collector diese abholt. Die aus dem SHM entnommenen Daten werden dort sofort wieder auf Null zurückgesetzt. Für den Datenaustausch soll das bewährte SNMP-Protokoll dienen, welches von vielen Applikation bereits unterstützt wird. Der zu integrierende SNMP-Server soll hierbei zwei Zugrifftypen erlauben. Einerseits die Zugriffe des data collectors, dem all die aufsummierten Datenbestände des speziellen Containers zur Verfügung gestellt werden sollen, andererseits Zugriffe beliebiger anderer SNMP-Clients auf selektiv geführte Datenbestände. Dieser spezielle Zugriff erlaubt beispielsweise MRTG[1] das Erstellen

[1] Multi Router Traffic Grapher - http://people.ee.ethz.ch/~etiker/webtools/mrtg/

von detaillierte Grafiken der Leitungsauslastung diverser Kundensegmente wie sie in Abb. 3.1 beispielhaft zu sehen ist.

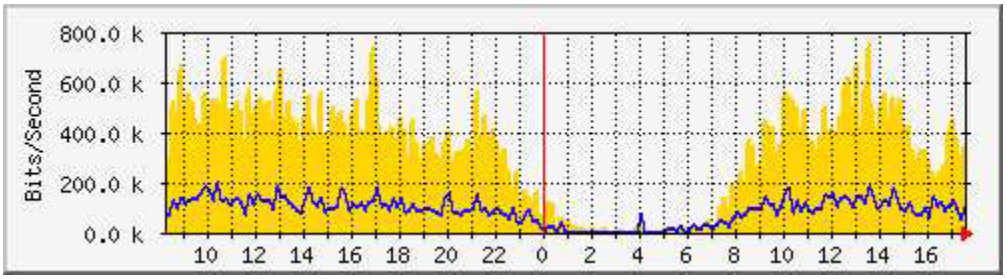


Abbildung 3.1: Beispielhafter MRTG Graph mit Hilfe der Paketverarbeitungsmaschine

Der `data collector` ist lediglich der Mittler zwischen den aufsummierten Daten der DAE und der Datenbank. Da wir laut Prämisse zeitgleich an verschiedenen Stellen im Netzwerk mittels PCE/DAE (sniffer Rechner) horchen dürfen, muss der data collector mit jeder DAE im Netz kommunizieren.
In Abb. 3.2 ist die Paketverarbeitungsmaschine skizziert.

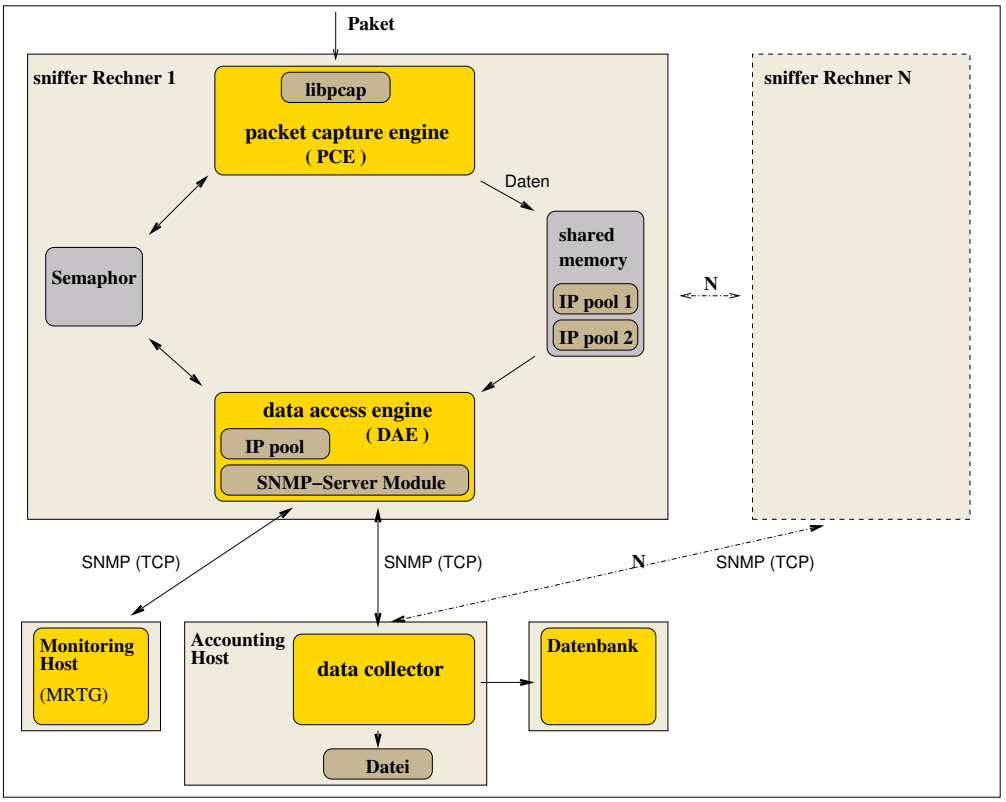


Abbildung 3.2: Struktureller Aufbau der Paketverarbeitungsmaschine

3.2 Datenmodellierung

Die Grundlage unserer Datenerfassung ist das struct `ipObject`, welches alle relevanten Informationen eines IP-Paketes beinhaltet.

```
typedef struct _ipObject
{
  struct _ipObject *nextInChain ;
  unsigned long adr;
  unsigned long in;
  unsigned long out;
  unsigned long udp;
  unsigned long http;
  unsigned long smtp;
  unsigned long ftp;
  unsigned long other;
} ipObject;
```

Die IP des Kunden wird hierbei in binärer Form in der `adr` und die Menge an übertragenen Bytes aufgeschlüsselt nach dem Serviceport[2] in den jeweiligen Unterkategorien (`http`, `ftp`, `other`, ..) abgelegt. Die Summe der Datenmenge, die unter dieser IP registriert worden ist, findet man in Abhängigkeit der Laufrichtung in `in` oder `out`. Das Feld `nextChain` ermöglicht eine Verkettung der ipObject und wird ausschließlich von dem Hash-Algorithmus verwendet. Die PCE sieht als einzige Komponente die Rohdaten des IP-Pakets und bildet daraus direkt diese ipObjects. Nachfolgende Komponente verwenden nur noch dieses struct.

Wichtige Daten die von der PCE als auch der DAE benötigt werden, sind am Anfang des SHM in dem struct `ipcDir` hinterlegt.

```
typedef struct _ipc_dir
{
  struct ether_addr etherIntern;
  struct ether_addr etherExtern;
  char cHostId[255] ;
  int cacheSampleThreshold;
  unsigned int iBucketPool1;
  unsigned int iBucketPool2;
  unsigned int iDAEPid ;
  unsigned int iPcePid ;
} ipcDir ;
```

Die Felder `etherIntern` und `etherExtern` legen die MAC-Adressen der Router Interfaces zwischen denen der Sniffer horcht fest. Mit Hilfe dieser ist die Fliessrichtung der Pakete feststellbar. `cHostId` beinhaltet den Namen der sniffer Rechner und wird zur Unterscheidung dieser verwendet. Die Felder `iBucketPool1`

[2] Der Serviceport ist im Header des TCP-/UDP Pakets zu finden

sowie `iBucketPool2` geben die Anzahl der ipObjects in den jeweiligen Datenbereichen der Hash-Tabelle an. Da wir für die Verarbeitung der ipObjects keine Verkettung verwenden, muss die Anzahl der insgesamt abgelegten ipObjects bekannt sein. Die Prozess ID (`PID`) der PCE und DAE sind ebenfalls abgelegt damit diese beiden Prozesse während der Ausführung kontrollieren können, ob ihr Partner noch läuft. Sollte dies nicht der Fall sein, werden die SHM Segmente ordnungsgemäß freigegeben und die Programmausführung gestoppt.

Da die Paketverarbeitungsmaschine mehreren tausend Pakete in der Sekunde kategorisieren soll, benötigt man hierzu effiziente Strukturen. Zu verarbeitende Pakete müssen entweder falls schon ein Paket mit gleicher Zugehörigkeit existiert aufsummiert werden, oder der Liste der ipObject angehängt werden. Untersuchungen (siehe [Sch02]) zeigten bei der Verwendung realer IP-Daten, dass Hash-Tabellen im Vergleich zu Skip-Lists und Splay-Trees besser für die Kategorisierung geeignet sind. Als Hash-Key dient uns die binäre Form der IP, welche in ipObject.adr abgelegt wurde. Zur Verwaltung der Hash-Tabellen verwenden wir das struct `ipDict`, welches ausschließlich im Userspace des betreffenden Programms liegt.

```
typedef struct _ipDict
{
  struct _ipDict  *next ;
  ipCacheId       id ;
  void *          *pObjPage ;
  ipObject        *pRoot ;
  hphTableRec     *pObjHash ;
  int   bucketCnt ;
} ipDict ;
```

Falls mehrere ipDict zusammengefasst werden sollen, kann man diese mittels `next` verketten. Die PCE verwendet zwei Hash-Tabellen und daher auch zwei ipDict, die miteinander verkettet sind. Das Feld `id` ermöglicht eine eindeutige Identifikation dieser. Die zu verwaltende Datenmenge wird ohne Lücken in dem `Datencontainer`, der ab dem Zeiger pObjPage anfängt hinterlegt. Die Anzahl an Objekten, die dort gespeichert sind werden im Feld `bucketCnt` hinterlegt. Der Zeiger `pObjHash` vom Typ `hphTableRec` beinhaltet eine Reihe von internen Informationen (zum Beispiel das Speeren des Zugriffs), die einen reibungslosen Betrieb der Hash-Tabelle ermöglichen. Die Hash-Tabelle ist Teil des Softwarepakets `libgeneric`, welche in der Softwareabteilung der “netplace Telematic GmbH“ entwickelt worden ist. Abb. 3.3 verdeutlicht den Aufbau des SHM.

Da diverse SNMP-Clients, wie beispielsweise der MRTG, lediglich absolute Werte verwalten können -in den Hash-Tabellen des SHM aber nur Differenzen stehen- wird zusätzlich noch das festes Array der Größe `NUM_IPOBJECTS` aus `MRTGipObjects` zur Speicherung der Daten eingesetzt.

```
MRTGipObjects MRTGipObjects[NUM_IPOBJECTS];
```

Das struct `MRTGipObjects` wird zudem um die Netzmaske `mask` erweitert, was uns eine Zusammenfassung von mehreren Kunden-IPs unter einem ipObject

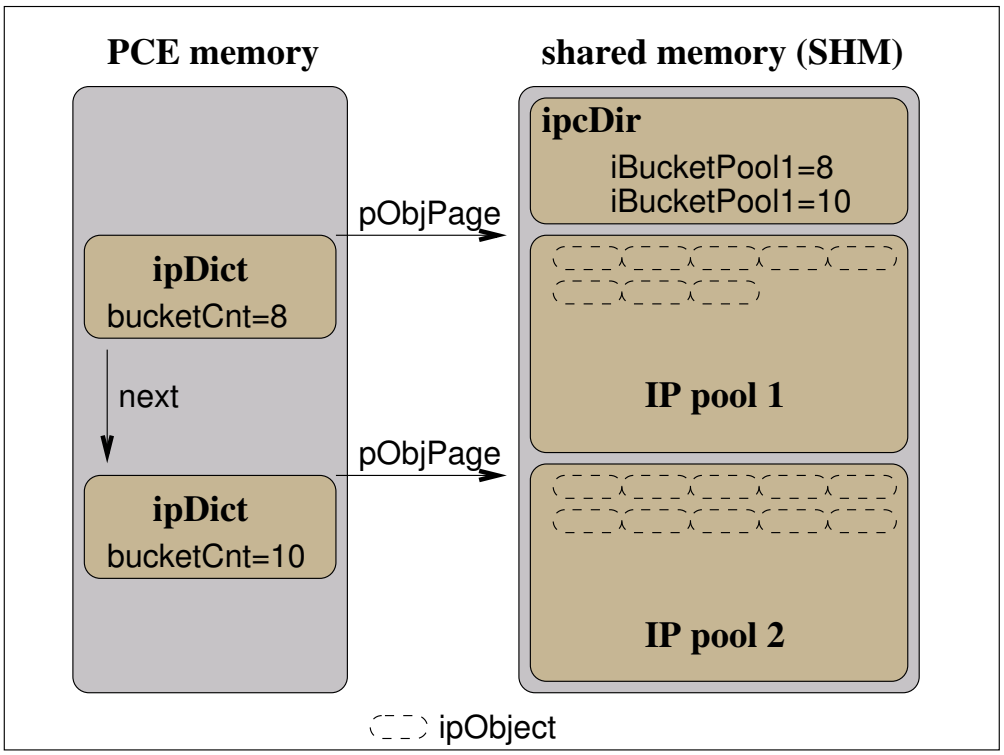


Abbildung 3.3: Aufbau des shared memory Bereichs mit der Hash-Tabelle

ermöglicht. Hierdurch können beispielsweise Flussstatistiken ganzer Netzsegmente aufgezeichnet werden.

```
typedef struct MRTGipObjects_ {
  ipObject ipObj;
  unsigned long mask;
} MRTGipObjects;
```

3.3 Synchronisation mittels Semaphoren

Der ausschließende Zugriff der DAE und PCE auf die zwei Datencontainer im SHM muss in unserem System zu jedem Zeitpunkt gewährleistet sein. Hierzu eignen sich Semaphoren am besten. Diese werden in unserem System zugleich auch zur Signalisierung der verschiedenen Zustände der Datencontainer verwendet. Signalen und Netzwerksockets zur Synchronisation können im unserem Fall nicht eingesetzt werden. Der wesentliche Nachteil an Signalen ist, dass diese die `select()` Aufrufe des integrierten SNMP-Servers terminieren würde und es daher zu nichtdeterministischen Verhalten führen könnte. Die Synchronisation mittels Netzwerksockets wiederum würde aufgrund des Overheads den Ablauf der PCE zu stark beeinflussen. Zudem sind beide Methoden verglichen mit der Semaphoransatz relativ aufwendig handzuhaben.

Jeder Datencontainer hat zwei unabhängige, binäre Zustände, die ihn beschreiben.

- Container ist verfügbar/in Verwendung. (USE)
- Container ist leer/voll. (FULL)

Beide Prozesse haben vollen Zugriff auf diese Variablen, dürfen diese aber nur in gewissen Situationen verändern. Sobald ein Prozess auf einen Container zugreift wird der Container von “verfügbar“ auf “in Verwendung“ geschaltet. Je nachdem wird danach der Zustand des Containers in “leer“ oder “voll“ verändert.
Die PCE kann einen Container laut unserer Definition zur Abspeicherung neuer Daten verwendet, wenn dieser verfügbar und leer ist. Sollte diese Bedingung nicht genügen, verwendet die PCE nach wie vor in den selben (“alten“) Container und wechselt diesen nicht. Die DAE prüft in regelmäßigen Abständen den Status der zwei Datencontainer und leert einen, falls dieser verfügbar und voll sein sollte.
Nach dem Start des Systems sind beide Container leer und verfügbar.

4 Implementierung des Systems

4.1 Allgemeines

Das Projekt wurde bei der "netplace Telematic GmbH" realisiert und verwendet daher für grundlegende Funktionen eine schon vor Jahren eigens entwickelte Bibliothek (libgeneric[1]). Aus dieser Bibliothek wurden Funktionen, die das shared memory Management sowie den Zugriff auf Datenbanken erleichtern, eingesetzt. Eine dort realisierte Hash-Tabelle wurde ebenfalls verwendet und modifiziert.
Für den Zugriff auf Netzwerkfunktionen des Betriebssystems wird die `libpcap` eingesetzt, welche uns zudem die Portierung auf andere Betriebssysteme erleichtert.
Aufgrund von diversen Probleme mit ucd-snmp, die sich vor allem bei großen Datenmengen bemerkbar machten, verwenden wir nun die Bibliotheken der `net-snmp` Projekts[2] um den SNMP-Server in der DAE zu realisieren.
Alle Komponenten verwenden `syslog()` um wichtige Informationen zu protokollieren. Zudem besteht die Möglichkeit mittels SIGHUP Signal eine extensive Protokollierung zu starten, die eine Fehlersuche erleichtern soll.
Im Folgenden werden Details zu den Implementierungen der Komponenten ausgeführt.

4.2 Packet Capture Engine

Wie bereits erwähnt soll die PCE eingehende Pakete, die vom Kernel in der `receive queue` zwischengelagert werden, möglichst ohne Unterbrechung herausnehmen und dem System in Form der ipObjects zur weiteren Verarbeitung zur Verfügung stellen.
Die PCE wird von der DAE mittels exec() gestartet, und bekommt von ihr über Kommandozeilenparameter die zuvor angelegte shared memory id (shmid) des SHM sowie dessen Größe mitgeteilt. Mit dieser Information hängt die PCE sich an den SHM und initialisiert diesen zunächst. Es werden im Userspace zwei Hash-Tabellen angelegt und die betreffenden zwei Datencontainer in den SHM gelegt. Durch die zwei Hash-Tabellen kann man gewährleisten, dass die PCE stets einen freien Container zur Ablage findet, wenn die DAE zeitgleich einen Container reserviert haben sollte. Die Datenübergabe ist durch

[1] diese wird zukünftig unter einer offenen Lizenz stehen
[2] NET-SNMP Project - http://net-snmp.sourceforge.net/

die beiden Container stets und ohne Unterbrechungen gewährleistet. Die Semaphoren für eben diese Synchronisation (siehe Kapitel 3.3) werden initial gesetzt und das Mithorchen gestartet. Es folgt eine Endlosschleife, welche nur im Fehlerfall durchbrochen wird. In dieser wird ein Paket nach dem anderen mit Hilfe der libpcap aus der receive queue entnommen. Im nächsten Schritt wird die Fliessrichtung des Pakets anhand der MAC-Adressen im Ethernet Header festgestellt. Die zu vergleichende MAC-Adresse ist hierbei statisch festgelegt, da sich diese solange die Router Hardware nicht ausgetauscht wird nicht ändert. Die Fliessrichtung ermöglicht eine exakte Zuweisung des Pakets zu den betreffenden Kunden. Bei Paketen, die vom Kundenrechner weg fließen wird die Source-IP und für eingehende Pakete die Destination-IP als Zuordnungsgrundlage verwendet. Im Folgenden bezeichne ich diese IP als Kunden-IP. Ein rudimentärer Filter stopt die weitere Bearbeitung von Paketen die außerhalb des zu untersuchenden Netzes liegen. Dieser Filter wird vor allem benötigt um Pakete mit falschen IP-Header Informationen frühzeitig auszusortieren damit unsere Datenbank nicht mit falschen Einträge überfüllt wird. Problematisch sind hierbei vor allem Viren und Trojaner, die die Source-IP auf beliebige Werte setzen um den Ursprung dieses Paketes und damit des befallenen Rechners zu verschleiern[3]. Die restlichen Pakete werden in Abhängigkeit der Kunden-IP aufsummiert, wobei aufgrund der Datenmenge effiziente Hash-Tabellen eingesetzt werden.
Die Aufgabe der PCE ist zudem den Wechsel des Datencontainers nach zuvor festgelegten Zeiteinheiten (Zyklus) durchzuführen, damit jeder Container abwechselnd geleert werden kann. Um nicht bei jedem Schleifendurchlauf die verstrichene Zeit mittels aufwendigen `time()` Aufrufen abfragen zu müssen, lösen wir das Problem auf indirekte Weise. Es wird jedes empfangene Pakete gezählt und überprüft ob die Summe einen adaptiven Schwellwert(`cacheSampleThreshold`) erreicht hat. Sollte dies der Fall sein zählen wir wieder von null an und wechseln den Container falls einer "frei" und "verfügbar" sein sollte. Der Schwellwert wird in Abhängigkeit der vergangen Zeit in diesem Zyklus angepasst.

$$neuer\_Threshold = alter\_Threshold \times \frac{SollZyklusdauer}{IstZyklusdauer}$$

Durch diese einfache Methode spart man sich viel Aufwand (vor allem Systemaufrufe), kann aber die Zyklusdauer mitsamt zeitgenauer Datenübergabe nicht mehr garantieren. Dies hat für unser System keine Auswirkungen, da jeder einzelne Datencontainer groß genug ist um Schwankungen der Zyklusdauer auszugleichen. Nach jedem Schleifendurchlauf prüft die PCE mittels `SIGNULL` Signal, ob die PID der DAE noch legitim ist und beendet ansonsten die Ausführung ordnungsgemäß indem der SHM wieder freigegeben wird. Tatsächlich wird bei SIGNULL nicht wirklich ein Signal geschickt und lässt sich daher problemlos zur Kontrolle anderer Prozesse einsetzen.

[3] der nimbda Wurm ist beispielhaft zu nennen

4.3 Data Access Engine

Die Hauptaufgabe der DAE ist die Datenübergabe zwischen den Datencontainer der PCE und den diversen Applikationen (beispielsweise dem data collector). Da diese Kommunikation asynchron ist, benötigt man zur effizienten Zwischenspeicherung der Daten eine eigene Hash-Tabelle mit Datencontainer. Diese liegen in Falle der DAE beide im Userspace der Anwendung. Die DAE ist derart implementiert, dass die Ausführung nie über einen längeren Zeitraum steht um zumindest in endlicher Zeit die Leerung einzelner Datencontainer der PCE zu gewährleisten. Der Datencontainer der DAE speichert nur Differenzwerte der übertragenen Datenmenge seit der letzten SNMP-Anfrage. Die DAE hat zudem die Aufgabe die an den data collector übergebenen Datensätze mit einem Zeitstempel zu versehn damit die Datenbank diese später richtig einordnen kann. Aufgrund dessen das das SNMP-Protokoll nur Strings übertragen kann, werden diese Datensätze gleich in fertige SQL-Anweisungen formatiert. Wie bereits erwähnt benötigen einige SNMP-Clients absolute Werte, weshalb Daten spezieller Kunden-IPs gegebenenfalls zusätzlich in die MRTGipObjects[] eingetragen werden. Überläufe dieser MRTGipObjects[] werden hierbei korrekt abgefangen.
Die DAE wird direkt über die Shell aufgerufen und führt daraufhin die nötigen Schritte aus um den Betrieb zu starten. Zunächst wird der SHM angelegt, initialisiert und die PCE als auch der SNMP-Server gestartet. In der folgenden Endlosschleife wird durch ein select() mit einem fünf Sekunden Timeout (`SNMP_TIMEOUT`) auf SNMP-Anfragen gelauscht. In dieser Zeit steht die Programmausführung und die DAE verbraucht keine Rechenzeit. Da wir ohnehin von einem asynchronen Ablauf ausgehen, wurden unsere Komponenten dementsprechend realisiert und es ergeben sich daher keine gravierenden Probleme. Sollte eine SNMP Anfrage eintreffen wird diese sofort abgehandelt. Im folgenden Schritt wird unter Verwendung der Semaphoren der Zustand der Datencontainer des SHM überprüft. Sollte ein Container “voll“ und “verfügbar“ sein, werden die dortigen Paketen verarbeitet und der Container anschließend geleert. Die Semaphore wird auf “leer“ zurückgesetzt und steht dadurch der PCE wieder zur Verfügung. Da die DAE keinen Zugriff auf die Hash-Tabelle der PCE hat aber die Daten hintereinander ohne Lücken im Datencontainer liegen, kann sie die einzelnen Pakete dennoch auslesen. Hierbei wird bis zu der Anzahl an Paketen des Containers (iBucketPool1 und iBucketPool2) Schritt für Schritt immer die Größe eines ipObject im SHM weiter gegangen. Am Ende eines Schleifendurchlaufs wird analog zur PCE mittels dem `SIGNULL` Signal überprüft, ob die PCE noch läuft und gegebenfalls die Programmausführung ordentlich gestoppt. Diese Endlosschleife ist im Schaubild 4.1 verdeutlicht.
Bei der Umsetzung des SNMP-Servers wurde ebenfalls auf Effizienz und Performanz geachtet. Daher war es nötig ein eigenes SNMP-Modul zu schreiben bei dem für jede Kunden-IP (ipObject) ein eigener Slot (SNMP `OID`) reserviert wird. Die einzelnen OIDs werden von keiner Anwendung direkt verwendet, sind aber aufgrund der SNMP-Spezifikationen zwingend notwendig. Der SNMP-Server unterscheidet also bei einer SNMP-Anfrage des data collectors nicht die ihm

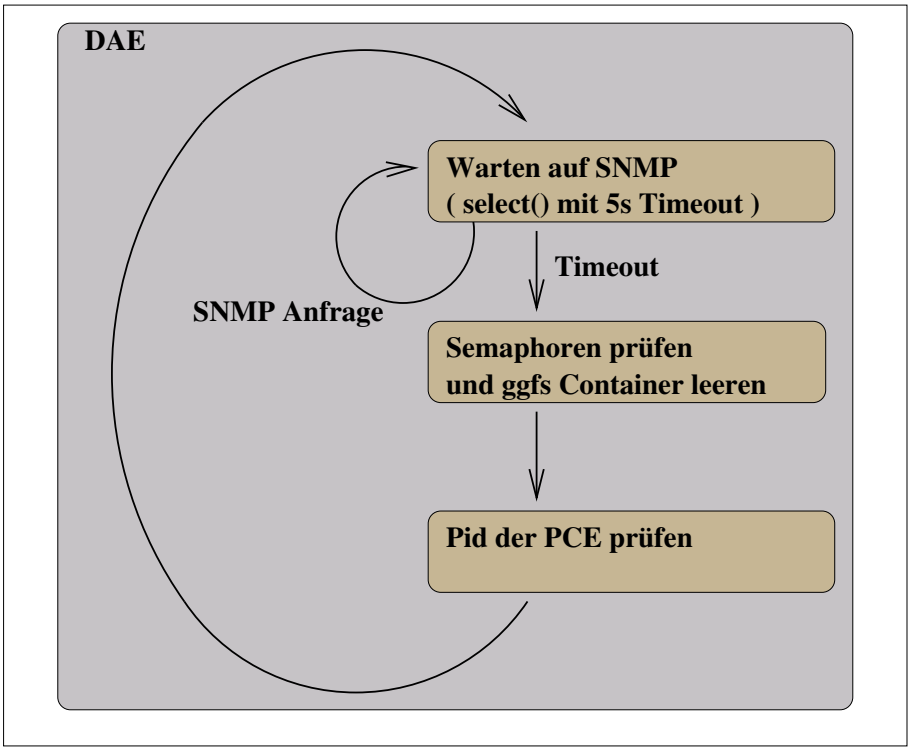


Abbildung 4.1: Ablaufdiagramm der DAE

mitgeteilten OIDs sondern gibt stets nur die nächsten -noch nicht verschickten- Datensätze weiter. Um zu verhindern, dass für jedes ipObject eine einzelne Rückantwort versendet wird, verwenden wir den speziellen Übertragungsmodus `SNMP_MSG_GETBULK`. Durch diesen können mehrerer ipObjects in einem Antwortpaket zusammengefasst werden. Problematisch hierbei war der Umstand, dass die Anzahl an ipObjects, die in einer solcher Antwort verschickt werden können, direkt von der Paketgröße der SNMP-Anfrage abhängt. Unser System wurde daraufhin so implementiert, dass in jeder SNMP-Antwort 400 ipObjects (`SNMP_MAXOBJS`) passen. Muss die DAE weniger oder mehr ipObjects verschicken, wird respektive der restliche Datenplatz der Antwort ausgenullt oder weitere SNMP-Antwortpakete mit den restlichen Daten verschickt. Zusätzlich hängen wir an das letzte SNMP-Paket die Gesamtzahl an übertragenen Datenmenge während des vergangenen Zyklus. Der data collector schickt neue SNMP-Anfragen und wertet diese zusätzlichen Informationen solange aus bis diese letzte Zusammenfassung empfangen worden ist.

4.4 Data Collector

Der data collector soll die Datenbestände der DAEs im System der Reihe nach mittels SNMP abfragen und die so empfangenen und bereits formatierten SQL-Anweisungen der Datenbank im Hintergrund zuschieben. Da eine weitere For-

matierung der mehreren tausend Anweisung nicht mehr nötig ist, wird der data collector ein wenig entlastet. Durch die Entlastung will man so gut es geht sicherstellen, dass der data collector seinen Aufgaben innerhalb eines Zyklus nachkommt. Ein Zyklus hat eine Zeitdauer von 60 Sekunden (`AC_COLLECT_CYCLE_TIMER`) in welcher möglichst alle DAEs abgearbeitet werden sollen. Erst wenn alle DAEs behandelt worden sind und diese Zeit verstrichen ist, wird ein neuer Zyklus gestartet. Schwankungen der tatsächlichen Zyklusdauer sind bei sehr großen Datenmengen[4] nicht auszuschließen, bedeuten aber lediglich dass der Zeitstempel der Datensätze in der Datenbank um eben diese Zeit differiert. Um die Ausführungszeit zusätzlich zu optimieren, bleibt die Verbindung zur Datenbank zu jeder Ausführungszeit bestehen. Man spart sich dadurch das erneute Anmelden in jedem Zyklus. Der Kommunikationsaufwand via SNMP wird wie bereits in Kapitel 4.3 erwähnt durch das Zusammenlegen mehrerer ipObjects in eine SNMP-Antwort reduziert.
Der data collector schreibt eine gemittelte Zusammenfassung der erfassten Datenmenge des letzten Zyklus, der letzten 5 Zyklen und der letzten 60 Zyklen in eine Datei. Diese Datei ermöglicht eine erste Fehleranalyse des Systems.
Zusätzlich besteht über das `SIGUSR1` Signal die Möglichkeit die Verbindung zur Datenbank zu kappen und die Ausführung des data collectors temporär zu stoppen. Da die von uns eingesetzte Datenbank Oracle 8.1.7i nur dann die neuen Datensätze in ihren Hintergrundspeicher schreibt wenn die Verbindung beendet ist, wird während des täglichen Backups der Datenbank der data collector kurzzeitig gestoppt.

[4] innerhalb von 60 Sekunden schafft man mindestens 50000 Datensätze

5 Zusammenfassung

Die im Rahmen des Systementwicklungsprojekts entwickelte Paketverarbeitungsmaschine für IP-Netzwerke wird bei der netplace Telematic GmbH eingesetzt und stellt die Grundlage zur Rechnungsstellung an den Kunden dar. Die Software kann an mehreren Stellen im Netzwerk eingesetzt werden und ermöglicht somit eine lückenlose Datenerfassung der Datenströme. Die einzelnen Komponenten wurde auf Effizienz getrimmt und nach mehrmonatigen Probedurchlauf ist von der Korrektheit der Datenerfassung auszugehen.
Da die Programmquellen der Paketverarbeitungsmaschine im Gegensatz zu kommerziellen Lösungen der netplace Telematic GmbH offen stehen, ist eine Erweiterung dieser um zusätzliche Features möglich. Beispielsweise könnte die Paketverarbeitungsmaschine um die Fähigkeit DOS[1]-Attacken zu erkennen und dementsprechend Abwehrmaßnahmen einzuleiten erweitert werden.

[1] Denial of Service

Abbildungsverzeichnis

Literaturverzeichnis

[Dan01] Dankwardt, Kevin: *Comparing real-time Linux alternatives.* http://www.linuxdevices.com/articles/AT4503827066.html, October 2001.

[Dyk01] Dykstra, Phil: *Protocol Overhead.* http://sd.wareonearth.com/ phil/net/overhead/, March 2001.

[Hed87] Hedrick, Charles L.: *General description of the TCP/IP protocols.* http://oac3.hsc.uth.tmc.edu/staff/snewton/tcp-tutorial/sec2.html, 1987.

[Ins01] Insolvibile, Gianluca: *Kernel Korner: The Linux Socket Filter: Sniffing Bytes over the Network.* http://www.linuxjournal.com/article.php?sid=4659, June 2001.

[Ins02a] Insolvibile, Gianluca: *Kernel Korner: Inside the Linux Packet Filter.* http://www.linuxjournal.com/article.php?sid=4852, February 2002.

[Ins02b] Insolvibile, Gianluca: *Kernel Korner: Inside the Linux Packet Filter, Part II.* http://www.linuxjournal.com/article.php?sid=5617, March 2002.

[Sch02] Schraffl, Tanja: *Entwurf eines "multiple probing" Systems zur Analyse und Messung des Datentransfervolumens in IP Netzwerken*, April 2002. Diplomarbeit.

[Shi] Shield, Dave: *Extending the UCD-SNMP agent.* http://www.csc.liv.ac.uk/ daves/Misc/UCD/guide.html.

[Tie] Tierney: *TCP Tuning Guide.* http://www-didc.lbl.gov/TCP-tuning/.

[Wil02] Williams, Clark: *Which is better – the preempt patch, or the low-latency patch? Both!* http://www.linuxdevices.com/articles/AT8906594941.html, March 2002.