Universität Potsdam

Mathematisch-Naturwissenschaftliche Fakultät

Institut für Informatik

Projektarbeit

tcpdump Analyse

Sebastian Menski (734272)

menski@uni-potsdam.de

Betreuer: Dipl.-Inf. Simon Kiertscher

Potsdam, 16. Mai 2016

Inhaltsverzeichnis

1 Einleitung

Diese Arbeit untersucht ob tcpdump[1] zum Monitoren von HTTP Traffic in einem Hochlastszenario geeignet ist. Der Use Case ist, dass tcpdump dazu eingesetzt werden soll HTTP Traffic zu überwachen und zu analysieren. Anhand der gewonnen Daten sollen Entscheidungen getroffen werden um eine Lastverteilungsinfrastruktur zu steuern. Um dies sinnvoll einzusetzen muss sichergestellt werden, dass das Monitoring alle oder mindestens den größten Teil der HTTP Requests aufzeichnen kann.

Um in diesem Kontext tcpdump zu evaluieren wird diese Arbeit zuerst die Grundlagen von tcpdump im Kapitel 2 erklärt. Dabei wird gezeigt wie tcpdump Pakete filtert und wo es dort zu Paketverlusten kommen kann. Anschließend werden in Kapitel 3 Arbeiten aufgeführt die sich bereits mit der Performance von tcpdump beschäftigt haben. Das Kapitel 4 beschreibt dann den Versuchsaufbau für diese Arbeit und was dabei die erwarteten Beobachtungen sind. Die Messergebnisse der Experimente werden in Kapitel 5 vorgestellt und ausgewertet. Das letzte Kapitel 6 gibt eine Zusammenfassung der Ergebnisse dieser Arbeit und eine Einschätzung ob tcpdump in dem beschrieben Use Case sinnvoll eingesetzt werden kann.

Die Beschreibungen und Messungen in dieser Arbeiten basieren alle auf der tcpdump Version 4.3.0 und der dazu gehörigen libpcap Version 1.3.0. Das Betriebssystem auf dem tcpdump lief war Centos 5 mit einem Kernel der Version 2.6.18.

Diese Arbeit befasst sich nur mit der Paketaufzeichnung und nicht mit der Weiterverarbeitung und Auswertung der aufgezeichneten Daten.

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

2 Grundlagen

Tcpdump[2] ist eine Software um Netzwerkpakete aufzuzeichnen. Tcpdump benutzt die Bibliothek libpcap, welche eine einheitliche API anbieten um Netzwerkpakete auf unterschiedlichen Betriebssystemen aus dem Netzwerkverkehr herauszufiltern. Ziel hierbei ist es Paket möglichst früh im Netzwerkstack zu filtern, wodurch nur Pakete von Interesse weiterverarbeitet werden müssen.

Diese Kapitel beschreibt wie tcpdump/libpcap in Linux Netzwerkpakete filtert. Dazu wird wie in [2, 3, 4] beschrieben der Verlauf eines eingehenden TCP Paketes durch den Linux Kernel betrachtet.

In Abbildung 1 sind die wichtigsten Teile des Linux Networkstacks dargestellt, welche bei dem Empfang eines TCP Paketes eine Rolle spielen. Im Userspace wird ein Webserver (nginx) betrieben, welcher über einen TCP Socket HTTP Requests empfängt. Parallel läuft tcpdump/libpcap welches einen speziellen `PF_PACKET` Socket nutzt um alle empfangen Pakete zu untersuchen (siehe Listing 1).

```
socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

Listing 1: Erzeugen eines `PF_PACKET` Sockets

Ein solcher Socket ermöglicht es Pakete direkt von der Netzwerkkarte zu erhalten ohne das die Pakete den normalen Netzwerkstack durchlaufen, dies ist wichtig für tcpdump da es das komplette Paket aufzeichnen will.

Erreicht ein TCP Paket die Netzwerkkarte wird es zuerst in eine Queue in der Netzwerkkarte (1) abgespeichert. Anschließend wird das Paket in die zuständige CPU Eingangsqueue übergeben (2). Danach übernimmt die Kernel Funktion `net_rx_action` die Weitergabe des Paketes. Dazu werden zwei Listen von Handler betrachtet. Als erstes wird das Paket an alle Handler übergeben, welche alle Pakete empfangen. Dazu zählt auch der `PF_PACKET` Socket von tcpdump/libpcap. Anschließend werden spezielle Handler für das Paket aufgerufen. In dem Fall eines TCP Paketes ist dies der IP Handler, welcher den IP Header liest, auswertet und entfernt. Anschließend wird das Paket an den TCP Handler übergeben. Dieser wertet den TCP Header aus und reicht das Paket an den zuständigen Socket weiter. Wenn ein Paket einen Socket erreicht wird es in dessen Eingangsbuffer (3) abgelegt.

Wie in Abbildung 1 zu sehen ist, gibt es mindestens drei Paketbuffer die ein Paket durchläuft. Wenn einer dieser Buffer voll ist, weil bereits zu viele Paket angenommen aber nicht verarbeitet wurden, wir ein neues Paket verworfen. Unter der Fragestellung unter welchen Umständen tcpdump Pakete verwirft, welcher allerdings die Anwendung (nginx) erreichen, sind die ersten beiden Buffer (NIC und CPU Queue) nicht interessant.

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

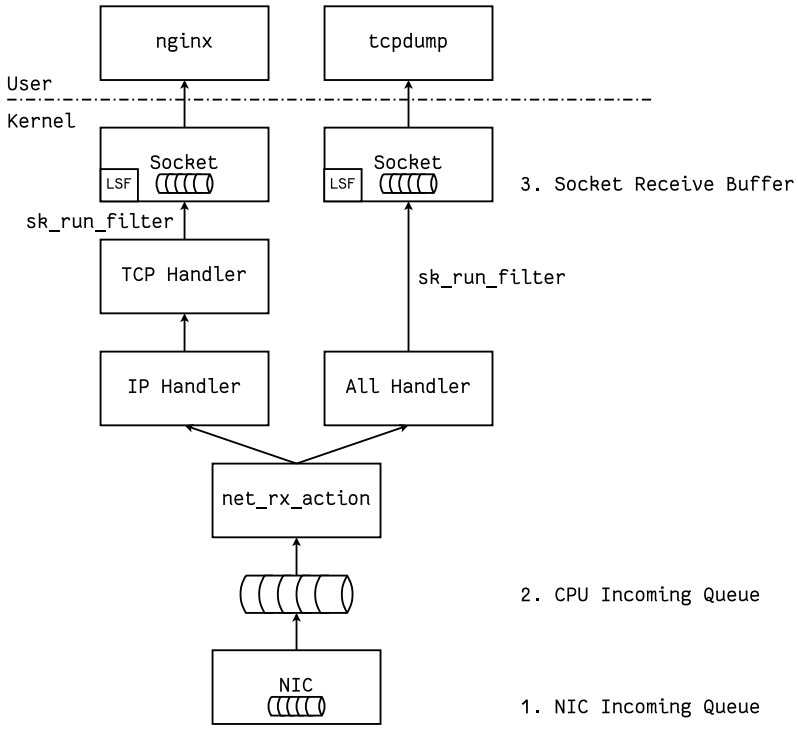


Abbildung 1: Schematischer Paketfluss beim Empfang eines TCP Paketes im Linux Kernel

Ein Paket, welches bereits an dieser Stelle verworfen wird, erreicht weder die Anwendung noch tcpdump. Insofern ist nur der dritte Buffer vom Socket interessant. An dieser Stelle hat das Paket bereits zwei unterschiedliche Wege durchlaufen. Dabei muss die Anzahl der Paket in beiden Sockets nicht identisch sein, da der `PF_PACKET` generell alle Pakete erhält. Dies bedeutet, wenn neben dem HTTP Verkehr noch weiterer Netzwerkverkehr existiert wird dieser ebenfalls an den tcpdump/libpcap Socket weitergeleitet.

Daher ist es nötig diesen Traffic möglichst frühzeitig zu filtern. Eine Filterung im Userspace würde das Problem nicht beheben, da immer noch alle Pakete erst einmal den Socket erreichen. Linux stellt für diese Zweck den Linux Socket Filter[3] (LSF) bereit. Diese API wurde mit Kernel 2.2 einführt und ermöglicht es ein Filter an einem Socket zu registrieren (siehe Listing 2).

```
setsockopt(socket, SOL_SOCKET, SO_ATTACH_FILTER, filter, sizeof(*filter));
```

Listing 2: Anhängen eines Filters an eine Socket

Der Filter ist ein Programm in der Assembler ähnlichen Sprache Berkley Paket Filter (BPF) [5]. Die Sprache umfasst einfache Vergleich- und Rechenoperation und arbeitet direkt auf den Daten des Netzwerkpaketes. Dadurch kann der Filter direkt auf einzelne Header Felder der verschieden Protokolle zugreifen. tcpdump/libpcap erlaubt es logische

[3] `https://www.kernel.org/doc/Documentation/networking/filter.txt`

Ausdrücke zum erstellen eines Filters in BPF umzuwandeln. Das Beispiel in Listing 3 zeigt den BPF Code für einen Filter der alle TCP Pakete für die Ziel-IP `10.3.9.21` und Ziel-Port `80` akzeptiert.

```
(000) ldh      [12]
(001) jeq      #0x800           jt 2    jf 12
(002) ld       [30]
(003) jeq      #0xa030915       jt 4    jf 12
(004) ldb      [23]
(005) jeq      #0x6             jt 6    jf 12
(006) ldh      [20]
(007) jset     #0x1fff          jt 12   jf 8
(008) ldxb     4*([14]&0xf)
(009) ldh      [x + 16]
(010) jeq      #0x50            jt 11   jf 12
(011) ret      #65535
(012) ret      #0
```

Listing 3: BPF for `tcpdump -d ip dst host 10.3.9.21 and tcp dst port 80`

Das BPF Programm lädt Daten aus dem Netzwerkpaket und vergleicht diese mit dem festgelegten Filter. Zuerst wird der EtherType aus dem Ethernet Header geladen (000) und verglichen ob es sich um ein IPv4 Paket (Type: 0x0800) handelt (001). Ist der Vergleich erfolgreich wird mit der nächsten Instruktion fortgefahren, ansonsten wird zu letzten Instruktion (012) gesprungen. Als zweites wird die Ziel-Adresse aus dem IP Header geladen (003) und mit der IP 10.3.9.21 (0xa030915) verglichen (004). Danach wird das Protokolfeld des IP Headers geladen (004) und überprüft, dass es sich um ein TCP Paket (0x6) handelt (005). In der folgende Instruktion wird sichergestellt, dass das IP Paket nicht fragmentiert wurde oder es sich um das erste Fragment handelt, also der IP Fragment Offset 0 ist (007). Anschließend wird die IP Headerlänge im Hilfsregister X gespeichert (008), dies ist nötig, da der IP Header durch Optionen eine variable Länge besitzt. Ein Zugriff auf den TCP Header ist nur mit dieser Information möglich. Als letztes wird der Ziel-Port aus dem TCP Header geladen (009) und mit Port 80 (0x50) verglichen (010). Wenn alle diese Tests erfolgreich waren wird mit Instruktion (011) das Paket akzeptiert. Ansonsten wird das Paket mit Instruktion (012) abgelehnt. Das BPF Programm gibt die Anzahl der Bytes zurück, welche von dem Paket weitergeleitet werden sollen. In libpcap ist der Standard dafür 65535 Bytes (siehe Instruktion 011), welches die maximale Größe eines IP Paketes entspricht. Der Rückgabewert 0 gibt an das kein Byte des Paketes weitergegeben werden soll (012).

Ist ein Filter für einen Socket registriert, wird dieser vom Handler durch die Kernel Funktion `sk_run_filter` ausgeführt, bevor das Paket an den Socket übergeben wird. Dadurch kann die Anzahl der Paket, welche den tcpdump/libpcap Socket erreichen effizient

reduziert werden.

Der tcpdump/libpcap Socket Buffer wird als Memory Mapped Packet Ring Buffer[4] angelegt. Dies ermöglicht es effizienter auf Paket im Userspace zu zugreifen. Dies verhindert, dass jedes Paket einzeln durch einen `recv` Aufruf empfangen werden muss. Die Größe des Buffers ist standardmäßig 2 MB. Der Buffer wird in Frames aufgeteilt, welche es ermöglichen das aufgezeichnete Paket und Metadaten zu speichern. Dabei richtet sich die Framegröße nach der Anzahl der Bytes die pro Paket aufgezeichnet werden.

Es zeigt sich, dass es mindestens drei Merkmale gibt, welche beeinflussen wieviel Paket tcpdump/libpcap aufzeichnen kann. Dies ist zu erst der Filter welcher möglichst viele Paket bereits im Kernel verwerfen sollte. Anschließend die Anzahl der Bytes die pro gefiltertem Paket weitergegeben werden und die Buffergröße die verwendet wird um die Paket vorzuhalten. Die Auswirkung dieser Aspekte soll in dieser Arbeit mit Hilfe von Messungen untersucht werden. Außerdem ist bereits aus Abbildung 1 ersichtlich das die Ausführung von tcpdump eine Auswirkung auf andere Services hat, da jedes Paket nun von einem zusätzlichen Handler betrachtet und gefiltert wird. Daher hat die Komplexität und Effektivität des tcpdump Filters einen direkten Einfluss auf die Geschwindigkeit mit der Paket verarbeitet werden.

[4] `https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt`

3 Verwandte Arbeiten

In [6, 7] wurde untersucht ob sich nicht spezialisierte Hardware zum Aufzeichnen von Paketen in Gigabit Netzwerken eignet. Es wurden 4 verschieden Systeme verglichen, welche sich nur in der verwendeten CPU und dem Betriebssystem unterschieden. Als CPUs wurden AMD Opteron und Intel Xeon verwendet. Die Betriebssysteme waren Linux 2.6.11 und FreeBSD 5.4. In den Messungen wurde der Linux Kernel Package Generator [9] genutzt um UDP Pakete zu generieren. Die Paketgrößen wurde anhand einer Verteilung generiert, welche aus einer 24h Aufzeichnung abgeleitet wurde. Die Ergebnisse der Experimente zeigten, dass mit steigender Paketrate die CPU Auslastung und der Paketverlust bei der Aufzeichnung steigt. Außerdem zeigte sich das die Kombination aus FreeBSD und AMD Opteron den geringsten Paketverlust aufzeigte. In einer anknüpfenden Arbeit [8] wurden Ansätze beschrieben wie man höhere Netzwerkarten wie 10-Gigabit Ethernet auf mehrere 1-Gigabit Interfaces verteilen kann. Dadurch kann der erhöhte Netzwerkverkehr immer noch effizient aufgezeichnet werden.

Den Einfluss von tcpdump auf die Leistung eines parallel laufenden Dienstes, wie zum Beispiel einem Webserver, wurde in [1] untersucht. Es zeigte sich eine starke Beeinträchtigung, wenn sich tcpdump und der Webserver eine CPU geteilt haben. Allerdings war auch bei getrennten CPUs die Leistung des Webservers beeinträchtigt.

Technologien wie Intel Data Plane Development Kit[5] (DPDK) and WireCAP[6] [10] untersuchen neue Ansätze um die Paketverarbeitung und -aufzeichnung zu optimieren. Dabei stehen Multi-Core Unterstützung und optimierte Speicherverwaltung im Vordergrund. Dabei können in Hochlastszenarien Verbesserungen zu normalen libpcap Anwendungen erzielt werden.

Die existierende Literatur zeigt, dass das Problem des Paketverlustes bei der Aufzeichnung von Netwerkverkehr unter hoher Last ein bekanntes Problem ist. Dabei haben viele Faktoren einen Einfluss auf die Paketaufzeichnung. Dies umfasst die verwendete Hardware, wie Netzwerkkarten, CPU und Festplatten, aber auch das verwendete Betriebssystem und Software. Da diese Arbeit sich explizit mit tcpdump/libpcap auf einem gegeben System befasst, wird im folgenden Kapitel ein Konzept vorgestellt um die Auswirkungen der in Kapitel 2 vorgestellten Aspekte von tcpdump/libpcap zu untersuchen.

[5] `http://dpdk.org/`
[6] `http://wirecap.fnal.gov/`

4 Konzept

Um die Leistung von tcpdump zur Aufzeichnung von HTTP Traffic zu zeigen wurden im Rahmen dieser Arbeit mehrere Messungen vorgenommen. Dieses Kapitel beschreibt das Konzept und den Aufbau dieser Messungen. Und begründet die Entscheidungen bezüglich der einzelne Komponenten.

Für die Messung standen zwei identische Systeme `ib5` und `ib6` zur Verfügung. Dabei wurde `ib5` als Benchmark Client und `ib6` als System unter Last (SUT) genutzt.

ib5/ib6

CPU Quad Core Intel Xeon 2.27 GHz

RAM 6 GB

OS CentOS 5.8

Kernel 2.6.18-308.el5

NIC Treiber bnx2 2.1.11

Als HTTP Benchmark wurde `wrk`[7] Version 4.0.2 auf `ib5` genutzt. Dieser Benchmark nutzt Multithreading und Event Loops um eine möglichst hohe Anzahl an HTTP Requests an das SUT zu schicken. Auf `ib6` wurde `nginx`[8] Version 1.9.15 als HTTP Server eingesetzt. Zur Aufzeichnung der Netzwerkpakete wurde `tcpdump`[9] Version 4.3.0 und `libpcap` Version 1.3.0 verwendet.

Ziel des Messaufbau war es eine möglichst hohe Anzahl an HTTP Requests zu erzeugen um eine hohe Last auf tcpdump zu generieren. Um dies zu erreichen wurden minimal HTTP Requests und Responses genutzt. Dazu wurde nginx so konfiguriert das es auf einen HTTP GET Requests direkt mit einem 204 No Content Response antwortet (siehe Listing 7 in Anhang A). Dies entspricht zwar keinem realen Szenario, allerdings kann eine reale Überlastsituation nicht mit der vorhanden Hardware simuliert werden. Es ist jedoch vorstellbar das die Anzahl der Pakete die durch diese Vereinfachung versendet wurden in einem realen Lastszenario erreicht werden können. Und eine geringere Last keine Probleme für tcpdump dargestellt hätten.

```
wrk -t 4 -c 1024 -d 5m http://ib6
```

Listing 4: Aufruf von wrk

Um die in Kapitel 2 aufgeführten Einstellungen von tcpdump zu untersuchen wurde insgesamt 6 Messreihen durchgeführt. Es wurde Untersucht wie sich tcpdump allgemein

[7] `https://github.com/wg/wrk`
[8] `http://nginx.org/`
[9] `http://www.tcpdump.org/`

	Name	snaplen	buffer	filter
1	no			—
2	default	65535	2048	ip dst host 172.16.0.26 and tcp dst port 80
3	snaplen	142	2048	ip dst host 172.16.0.26 and tcp dst port 80
4	buffer	65535	4096	ip dst host 172.16.0.26 and tcp dst port 80
5	snaplen+buffer	142	4096	ip dst host 172.16.0.26 and tcp dst port 80
6	filter	142	4096	ip dst host 172.16.0.26 and tcp dst port 80 and 'tcp[((tcp[12:1] & 0xf0) » 2):4] = 0x47455420'

Tabelle 1: Konfiguration von tcpdump für Messszenarien

auf die Performance des Webservers auswirkt. Und wie sich die Parameter Größe der aufgezeichneten Pakete (snaplen), Größte des Paketbuffers (buffer) und der verwendete Paketfilter (filter) auf tcpdump auswirken. Bei allen Messungen war die nginx Konfiguration gleich (siehe Listing 7 in Anhang A). Ebenfalls wurde wrk immer wie in Listing 4 gezeigt aufgerufen. Jede Messung war 5 Minuten lang und benutzt 4 Threads und 1024 Verbindungen. Die Konfiguration von tcpdump wurde für jede Messung angepasst (siehe Tabelle 1). Im Folgenden sollen die einzelnen Szenarien beschrieben werden.

no Die erste Messreihe ist eine Kontrollmessung ohne tcpdump. Diese Messung soll die maximale Anzahl der Requests in der Messumgebung ermitteln. Anhand dieser Messung können dann Aussagen über den Einfluss von tcpdump auf die Leistung des HTTP Servers getroffen werden.

default Bei dieser Messungen wurde die Standardparameter von tcpdump verwendet. Dabei ist die snaplen 65535 Bytes und der buffer 2048 KBytes. Der Filter beschreibt alle TCP Pakete für die Ziel-Adresse `172.16.0.26` und dem Ziel-Port `80`. Dies umfasst alle eingehenden TCP Pakete für den Webserver.

snaplen In diesem Szenario wurden die snaplen auf 142 Bytes reduziert. Die ersten 142 Bytes eines TCP Paketes reichen aus um zu analysieren ob es sich um ein HTTP Paket handelt. Dabei ergibt sich 142 Bytes aus der Mac Header Größe (14 Bytes), der maximalen IP Header Größe (60 Bytes), der maximalen TCP Header Größe (60 Bytes) und den ersten 8 Bytes der TCP Payload (HTTP Methode). Diese Aufzeichnungsgröße ist ausreichend um zu analysieren wieviel HTTP Pakete den Webserver erreichen, was dem untersuchten Use Case entspricht. Da für den IP und TCP Header die maximalen Größen angenommen wurden, werden vermutlich die meisten aufgezeichneten Pakete noch weitere Informationen

wie den Request Pfad enthalten.

buffer In diesem Szenario wurde die Buffergröße auf 4096 Bytes verdoppelt, dies sollte es tcpdump ermöglichen mehr Pakete für die Weiterverarbeitung vorzuhalten.

snaplen+buffer Diese Szenario kombiniert die Szenarien `snaplen` und `buffer` um die Anzahl der Pakete welche in den tpcdump Buffer passen zu maximieren.

filter Das letzte Szenario soll die Auswirkung des gewählten Filters untersuchen. Dazu wurde der Filter noch einmal verfeinert. Der Filter für Ziel-IP und Ziel-Port akzeptiert sowohl HTTP Requests als auch TCP Pakete welche zum Verbingungsaufbau und -abbau gesendet werden. Um diese bereits im Kernel herauszufiltern wurde der Filter um den Ausdruck `tcp[((tcp[12:1] & 0xf0) » 2):4] = 0x47455420` erweitert. Dieser Filter testet ob die ersten 4 Bytes der TCP Payload „`GET` “ entsprechen. Der Filter akzeptiert also nur Pakete die GET HTTP Requests sind. Außerdem wurden die snaplen und buffer Größe vom `snaplen+buffer` Szenario übernommen.

Jede Messung wurde 7 mal wiederholt. Dabei wurden die Ausgaben von wrk und tcpdump aufgezeichnet und ausgewertet. Die folgenden Metriken wurden dann untersucht:

1. Anzahl der gesendeten HTTP Requests (wrk)
2. Anzahl der gefilterten Netzwerkpakete (tcpdump)
3. Anzahl der verlorenen Netzwerkpakete (tcpdump)

Es wurde erwartet, dass anhand der Ergebnisse gezeigt werden kann, dass tcpdump eine Auswirkung auf die Anzahl der gesendeten HTTP Requests hat. Außerdem das es bereits in diesem einfach Messaufbau mit nur einem Client zu Paketverlusten kommt. Diese jedoch durch eine Anpassung der tcpdump Parameter snaplen und buffer reduziert werden können. Weiterhin ist die Vermutung, dass ein sehr spezifischer Filter eine weitere Verbesserung ermöglicht. Das folgende Kapitel 5 beschreibt die Ergebnisse der einzelnen Messungen und wertet diese aus.

5 Messungen

Dieses Kapitel stellt die Messergebnisse vor und analysiert diese. Vor der Messungen wurden keine Änderungen an den Systemkonfigurationen durchgeführt, außer die maximale Anzahl an offenen Dateien auf 4096 erhöht. Eine Auflistung der restlichen Limits und die Konfiguration der Netzwerkkarte sind in Anhang A in den Listings 8 und 9 zu finden. Die Messungen wurden durch ein Skript automatisiert, welches im Anhang B Listing 10 gezeigt ist.

Für jeden Messungen wurde die Ausgabe von wrk und tcpdump aufgezeichnet. Aus den Ausgaben wurden anschließend die Messwerte extrahiert. Am Ende eines Benchmarks gibt wrk eine Übersicht der gesendeten Requests aus (siehe Listing 5). Für die Analyse wurde die Gesamtzahl der gesendeten Requests verwendet und die Anzahl der Requests pro Sekunde.

```
Running 5m test @ http://ib6
  4 threads and 1024 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     5.32ms   13.26ms 610.71ms   92.91%
    Req/Sec   101.27k    15.99k  163.25k    58.68%
  120910283 requests in 5.00m, 12.38GB read
Requests/sec: 403008.38
Transfer/sec:     42.26MB
```

Listing 5: Ausgabe von wrk

Die aufgezeichneten Pakete von tcpdump wurden in eine Datei geschrieben (`-w`), um keine Mehraufwand durch die Ausgabe zu erzeugen. Diese Datei könnte von einem anderen Programm ausgewertet werden. Wird tcpdump beendet gibt es eine Übersicht der empfangen Pakete aus (siehe Listing 6). Für die Auswertung wurden die Werte „ packets received by filter“ und „packets dropped by kernel“ verwendet. Dabei gibt „ packets received by filter“ die Anzahl der Pakete an, die vom Filter akzeptiert wurden. Und „packets dropped by kernel“ die Anzahl der Pakete, welche zwar vom Filter akzeptiert wurden allerdings nicht von tcpdump aufgezeichnet werden konnten.

```
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size 65535
    bytes
94084513 packets captured
95945157 packets received by filter
1860644 packets dropped by kernel
```

Listing 6: Ausgabe von tcpdump

Die gesamten Messergebnisse sind in Anhang C aufgeführt. Zur Auswertung wurden jeweils der Median der Messwerte verwendet. Die Tabelle 2 zeigt für jedes Szenario den Median der gesendeten HTTP Pakete (Requests), die Requests pro Sekunde, die Anzahl

	Requests	**Requests/s**	**Filtered**	**Dropped**	**Dropped %**
no	120069354	400217,48			
default	93728797	312360,77	97520639	7446903	7,662 %
snaplen	94144165	313743,49	97952561	1634344	1,661 %
buffer	94197335	313920,03	98005865	1860171	1,898 %
snaplen+buffer	93635020	312040,81	97422724	142459	0,145 %
filter	94126289	313676,56	94128018	3613	0,004 %

Tabelle 2: Median der Messwerte für alle Szenarien

der von tcpdump gefilterten Pakete (Filtered), der von tcpdump verloren Pakete (Dropped) und dem prozentualen Anteil der verlorenen Pakete (Dropped %). Während keiner Messung wurden Fehler von wrk gemeldet, das heißt alle Requests konnten erfolgreich von nginx beantwortet werden. Es ist zu beachten, dass von tcpdump auch Pakete aufgezeichnet wurden, welche nicht mehr von wrk beachtet wurden. Da bei handelt es sich um Pakete die nicht mehr in dem festgelegten Zeitraum beantwortet wurden. Daher ist die zum Beispiel die Anzahl der gefilterten Pakete im Szenarion `filter` höher als die Anzahl der HTTP Requests. Allerdings ist diese Abweichung so gering, dass sie keine Auswirkung auf die Messergebnisse hat und vernachlässigbar ist.

Wie in Abbildung 3 zu erkennen ist nimmt die Anzahl der gesendeten HTTP Requests von wrk stark ab sobald tcpdump parallel zum Webserver betrieben wird. Dabei kommt es zu eine Reduzierung der Requests um circa 25 % von rund 120 Million auf rund 90 Million Requests. Dies kann durch die erhöhte Last auf dem ib6 Server erklärt werden. Zum einen werden nun alle Requests von zwei Sockets verarbeiten. Außerdem teilen sich der Webserver und tcpdump nun die verfügbaren Rechenkapazität auf dem Server. Weiterhin ist zu erkennen, dass dieser Einfluss nahe zu konstant für alle Messungen mit tcpdump ist.

In Abbildung 3a ist die Anzahl der von tcpdump verloren Pakete gezeigt. Zusätzlich zeigt 3b den prozentualen Anteil der verloren Paket von den empfangen Paketen. Es zeigt sich das der Paketverlust im Szenario `default` am Höchsten ist. Und das durch die Anpassung der tcpdump Parameter eine starke Verbesserung erzielt werden kann. Es scheint das die Reduzierung der aufgezeichneten Paketgröße einen größeren Einfluss hat als eine Erhöhung des Empfangsbuffers. Dies kann dadurch erklärt werden, dass die Paketgröße sehr stark reduziert wurde, von 65535 auf 142 Bytes. Zwar wurden keine Pakete versendet, welche 65535 Bytes groß sind, allerdings entscheidet diese Größe wie libpcap den Ring Empfangsbuffer aufteilt. Dabei geht es von der größt möglichsten Paketgröße aus. Mit einer geringen Paketgrößte kann somit der vorhanden Buffer besser genutzt werden. Die Kombination der beiden Parameter führte zu eine Reduzierung der verloren Pakete von

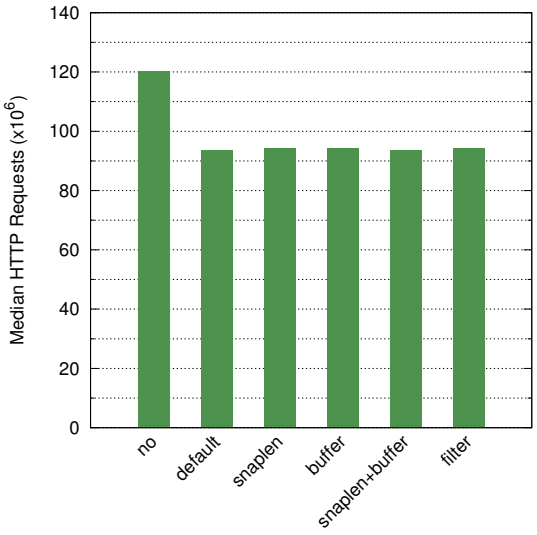
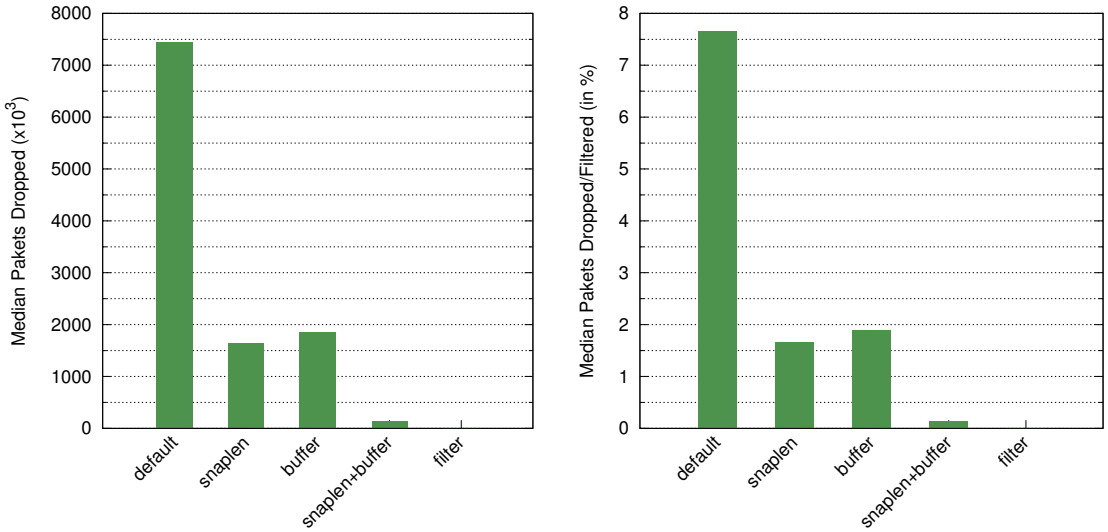


Abbildung 2: Anzahl der von wrk gesendeten HTTP Requests



(a) Anzahl der Pakete die vor tcpdump verloren wurden

(b) Prozentuale Anzahl der Pakete die vor tcpdump verloren wurden

Abbildung 3: Paketverlust bevor tcpdump die Pakete auswerten konnte

rund 98 % im Vergleich zu dem `default` Szenario. So wurden im Szenario `snaplen+buffer` nur noch 0.145 % der empfangen Paket verloren. Im Letzten Szenario `filter` konnte noch einmal eine Verbesserung durch den optimierten Filter erzielt werden, hierbei sank die Zahl der verloren Pakete auf 0.004 %. Somit konnten fast alle Pakete bei einer Last von rund 300.000 Requests pro Sekunde aufgezeichnet werden.

6 Zusammenfassung

In dieser Arbeit wurde untersucht, wie tcpdump/libpcap Netzwerkpakete filtert und aufzeichnet. Dazu wurde zu erst gezeigt, wie ein TCP Paket durch den Linux Netzwerkstack verarbeitet wird. Dabei wurden die möglichen Ursachen für den Paketverlust von tcpdump erklärt. Anschließend wurde ein Messkonzept erstellt, um die Auswirkung dieser Punkte zu untersuchen. Die daraus resultierenden Messungen bestätigten die Erwartung, dass diese Parameter entscheidend für die Anzahl der verlorenen Pakete sind.

Es zeigte sich das es sehr wahrscheinlich zu Paketverlusten kommen kann, wenn tcpdump zum Aufzeichnen von sehr hohem Netzwerkverkehr genutzt werden soll. Allerdings kann diese Problem durch gezielte Auswahl der tcpdump Einstellungen reduziert werden. Wie spezifisch die Einstellungen gewählt werden können hängt vom jeweiligen Use Case ab. So konnte für den hier diskutierten Einsatzzweck, der HTTP Lastverteilung, sehr spezifische Einstellungen vorgenommen werden. Soll jedoch zum Beispiel der komplette TCP Verkehr von mehreren Anwendungen aufgezeichnet werden, zum Beispiel für Intrusion Detection, dann muss ein recht grober Filter gewählt werden und ein großer Teil des Paketes aufgezeichnet werden.

Zusammenfassend konnte gezeigt werden, dass tcpdump zum Aufzeichnen von HTTP Requests genutzt werden kann, sofern die Einstellungen auf diesen speziellen Einsatzzweck optimiert werden. Außerdem sollte tcpdump nicht parallel auf dem System betrieben wird, welches unter Last steht, da es einen Einfluss auf die Leistung des Systems hat. Daher wäre es zu empfehlen das tcpdump Monitoring auf einem dedizierten System durchzuführen. Dies ist auch deshalb zu empfehlen, weil eine Weiterverarbeitung der tcpdump Aufzeichnung nötig ist, welche in dieser Arbeit nicht betrachtet wurde.

Literatur

[1] Chen, Chao-Chih, Yung Ryn Choe, Chen-Nee Chuah und Prasant Mohapatra: *Experimental Evaluation of the Impact of Packet Capturing Tools for Web Services.*
In: *Proceedings of the Global Communications Conference, GLOBECOM 2011, 5-9 December 2011, Houston, Texas, USA*, Seiten 1–5. IEEE, 2011.

[2] Insolvibile, Gianluca: *Kernel Korner: Linux Socket Filter: Sniffing Bytes over the Network.*
86:24, 26, 28, 30–31, Juni 2001.

[3] Insolvibile, Gianluca: *Kernel Korner: Inside the Linux Packet Filter.*
94:14, 16–18, Februar 2002.

[4] Insolvibile, Gianluca: *Kernel Korner: Inside the Linux Packet Filter, Part II.*
95:20–21, 23–25, März 2002.

[5] McCanne, Steven und Van Jacobson: *The BSD Packet Filter: A New Architecture for User-level Packet Capture.*
In: *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, Seiten 259–270. USENIX Association, 1993.

[6] Schneider, Fabian: *Performance evaluation of packet capturing systems for high-speed networks.*
Diplomarbeit, Technische Universität München, Munich, Germany, November 2005.

[7] Schneider, Fabian und Jörg Wallerich: *Performance evaluation of packet capturing systems for high-speed networks.*
In: Diaz, Michel, Arturo Azcorra, Philippe Owezarski und Serge Fdida (Herausgeber): *Proceedings of the 2005 ACM Conference on Emerging Network Experiment and Technology, CoNEXT 2005, Toulouse, France, October 24-27, 2005*, Seiten 284–285. ACM, 2005.

[8] Schneider, Fabian, Jörg Wallerich und Anja Feldmann: *Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware.*
In: Uhlig, Steve, Konstantina Papagiannaki und Olivier Bonaventure (Herausgeber): *Passive and Active Network Measurement, 8th Internatinoal Conference, PAM 2007, Louvain-la-neuve, Belgium, April 5-6, 2007, Proceedings*, Band 4427 der Reihe *Lecture Notes in Computer Science*, Seiten 207–217. Springer, 2007.

[9] Turull, Daniel, Peter Sjödin und Robert Olsson: *Pktgen: Measuring performance on high speed networks.*

Computer Communications, 82:39–48, 2016.

[10] Wu, Wenji und Phil DeMar: *WireCAP: a novel packet capture engine for commodity NICs in high-speed networks.*
In: Williamson, Carey, Aditya Akella und Nina Taft (Herausgeber): *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014*, Seiten 395–406. ACM, 2014.

A Konfigurationen

```
user root;
worker_processes  4;

events {
    use epoll;
    worker_connections  1024;
    multi_accept on;
}

error_log /dev/null crit;

http {
    access_log off;

    server {
        listen 80;

        location = / {
            return 204;
        }
    }
}
```

Listing 7: Konfiguration von nginx auf ib6

```
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 55296
max locked memory       (kbytes, -l) 32
max memory size         (kbytes, -m) unlimited
open files                      (-n) 4096
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 10240
cpu time               (seconds, -t) unlimited
max user processes              (-u) 55296
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Listing 8: Limits von ib5 und ib6 (ulimit)

```
driver: bnx2
version: 2.1.11
firmware-version: 5.0.13 bc 5.0.11 NCSI 2.0.5
bus-info: 0000:01:00.1

Ring parameters for eth1:
Pre-set maximums:
RX:     2040
RX Mini:    0
RX Jumbo:   8160
TX:     255
Current hardware settings:
RX:     255
RX Mini:    0
RX Jumbo:   0
TX:     255

Offload parameters for eth1:
rx-checksumming: on
tx-checksumming: on
scatter-gather: on
tcp segmentation offload: on
udp fragmentation offload: off
generic segmentation offload: off
generic-receive-offload: off
```

Listing 9: Konfiguration der Netzwerkkarten von ib5 und ib6 (ethtool)

B Messskript

```
#!/bin/bash

set -x
set -e

iter=${1:?Iterations required}
offset=${2:?Offset required}

NGINX=/root/menski/nginx-1.9.15/build/sbin/nginx
TCP_DUMP=/root/menski/tcpdump-4.3.0/build/sbin/tcpdump
WRK=/root/menski/wrk-4.0.2/wrk
LOGS=/root/menski/logs
DUMP=/root/menski/dump.pcap
DURATION=5m
DELAY=1m
THREADS=4
CONNECTIONS=1024

function start_nginx {
    ssh ib6 "$NGINX"
}

function stop_nginx {
    ssh ib6 "$NGINX -s quit"
}

function run_wrk {
    local log_file="${LOGS}/wrk-${1}-${2}.log"
    $WRK -t $THREADS -c $CONNECTIONS -d $DURATION http://ib6 > $log_file
    cat $log_file
}

function start_tcpdump {
    ssh ib6 "$TCP_DUMP -i eth1 -w $DUMP $3 &> ${LOGS}/tcpdump-${1}-${2}.log
        &"
}

function stop_tcpdump {
    ssh ib6 "pkill tcpdump"
}

function rm_dump {
    ssh ib6 "rm -f $DUMP"
}
```

```
44
45 for n in $(seq 1 $iter); do
46     let i=$n+$offset
47
48     bench="no"
49     start_nginx
50     sleep 5s
51     run_wrk $bench $i
52     sleep 5s
53     stop_nginx
54     sleep $DELAY
55
56     bench="default"
57     start_nginx
58     start_tcpdump $bench $i "-s 65535 -B 2048 ip dst host 172.16.0.26 and
           tcp dst port 80"
59     sleep 5s
60     run_wrk $bench $i
61     sleep 5s
62     stop_tcpdump
63     stop_nginx
64     rm_dump
65     sleep $DELAY
66
67     bench="snaplen"
68     start_nginx
69     start_tcpdump $bench $i "-s 142 -B 2048 ip dst host 172.16.0.26 and tcp
           dst port 80"
70     sleep 5s
71     run_wrk $bench $i
72     sleep 5s
73     stop_tcpdump
74     stop_nginx
75     rm_dump
76     sleep $DELAY
77
78     bench="buffer"
79     start_nginx
80     start_tcpdump $bench $i "-s 65535 -B 4096 ip dst host 172.16.0.26 and
           tcp dst port 80"
81     sleep 5s
82     run_wrk $bench $i
83     sleep 5s
84     stop_tcpdump
85     stop_nginx
86     rm_dump
```

```
87      sleep $DELAY
88
89      bench="snaplen+buffer"
90      start_nginx
91      start_tcpdump $bench $i "-s 142 -B 4096 ip dst host 172.16.0.26 and tcp
            dst port 80"
92      sleep 5s
93      run_wrk $bench $i
94      sleep 5s
95      stop_tcpdump
96      stop_nginx
97      rm_dump
98      sleep $DELAY
99
100     bench="filter"
101     start_nginx
102     start_tcpdump $bench $i "-s 142 -B 4096 ip dst host 172.16.0.26 and tcp
            dst port 80 and 'tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x47455420'"
103     sleep 5s
104     run_wrk $bench $i
105     sleep 5s
106     stop_tcpdump
107     stop_nginx
108     rm_dump
109     sleep $DELAY
110 done
```

Listing 10: Skript zum Automatisieren der Messungen

C Messwerte

	Requests	Requests/s	Filtered	Dropped	Dropped %
1	119903581	399653,93	-	-	-
2	120376044	401229,17	-	-	-
3	120268758	400880,73	-	-	-
4	119940796	399784,83	-	-	-
5	120069354	400217,48	-	-	-
6	119927287	399735,63	-	-	-
7	120273002	400894,69	-	-	-

Tabelle 3: Messwerte für das Szenario `no`

	Requests	Requests/s	Filtered	Dropped	Dropped %
1	93839932	312735,33	97637565	8000042	8,194 %
2	94485540	314891,25	98306066	8334649	8,478 %
3	93728797	312360,77	97520639	7030017	7,209 %
4	93412256	311292,88	97190374	7446903	7,662 %
5	92544907	308420,4	96289568	7288170	7,569 %
6	94540226	315071,23	98364247	8474911	8,616 %
7	93161788	310471,57	96933477	6717941	6,930 %

Tabelle 4: Messwerte für das Szenario `default`

	Requests	Requests/s	Filtered	Dropped	Dropped %
1	93361461	311133,8	97139116	1638937	1,687 %
2	94297388	314260,2	98111218	1142023	1,164 %
3	93455367	311449,63	97234642	2256178	2,320 %
4	94018148	313314,12	97820731	1224453	1,252 %
5	94144165	313743,49	97952561	1806995	1,845 %
6	94576465	315186,97	98400914	1634344	1,661 %
7	94243027	314073,82	98053289	1608101	1,640 %

Tabelle 5: Messwerte für das Szenario `snaplen`

	Requests	Requests/s	Filtered	Dropped	Dropped %
1	94479121	314868,26	98301445	3138511	3,193 %
2	93174474	310509,97	96945422	1464062	1,510 %
3	94197335	313920,03	98005865	1860171	1,898 %
4	94523845	315006,2	98346962	2196551	2,233 %
5	94215312	313968,29	98025862	1768006	1,804 %
6	93714952	312318,19	97506245	1445038	1,482 %
7	93451164	311448,09	97232458	2033839	2,092 %

Tabelle 6: Messwerte für das Szenario `buffer`

	Requests	Requests/s	Filtered	Dropped	Dropped %
1	93514281	311646,27	97298231	102277	0,105 %
2	94436463	314707,05	98255678	142459	0,145 %
3	93635020	312040,81	97422724	460590	0,473 %
4	93659731	312103,79	97448350	14219	0,015 %
5	94041647	313395,65	97844574	279299	0,285 %
6	93608679	311969,3	97396242	330139	0,339 %
7	92847251	309425,47	96604390	0	0,000 %

Tabelle 7: Messwerte für das Szenario `snaplen+buffer`

	Requests	Requests/s	Filtered	Dropped	Dropped %
1	93881869	312874,77	93883778	4893	0,005 %
2	93082658	310188,31	93084571	3613	0,004 %
3	94126289	313676,56	94128018	20014	0,021 %
4	94240703	314049	94242269	2534	0,003 %
5	92784745	309215,6	92786720	0	0,000 %
6	94166541	313826,58	94168634	23905	0,025 %
7	94220654	314003,73	94222585	2806	0,003 %

Tabelle 8: Messwerte für das Szenario `filter`