Projektdokumentation

Einsatz von GPS-Empfängern zur Bestimmung von One-Way-Delays in einem Netzwerk

David Wittwer

Universität Bern
Institut für Informatik und angewandte Mathematik (IAM)
Rechnernetze und verteilte Systeme (rvs)
Bern, Schweiz

Überblick

Die Bestimmung von Einweg-Verzögerungen ist ein Standardproblem in der Netzwerktechnik. Es liegt in der Natur von solchen „One-Way-Delays", dass ihre genaue Bestimmung nur mit exakt synchronisierten Uhren in den beteiligten Hosts zu bewerkstelligen ist. GPS und der Einsatz entsprechender Empfangsgeräte erlaubt es, die Uhren der Hosts auf einige Mikrosekunden genau zu setzen, was für die One-Way-Delay-Messung ausreicht. Andere Methoden zur Synchronisation, wie z.B. das Paket-basierte Network Time Protocol (NTP), bieten in der Regel keine ausreichende Genauigkeit.

Dieses Dokument zeigt einen möglichen Einsatz von GPS-Empfängern zur Bestimmung von Einweg-Verzögerungen zwischen zwei Linux-Hosts in einem Netzwerk. Ziel der Arbeit ist es, ein Höchstmass an Genauigkeit und Synchronisation zwischen Hosts herzustellen. Als Proof-of-Concept wurde weiter ein kleiner Demonstrator implementiert, welcher die Bestimmung der One-Way-Delays auch tatsächlich ermöglicht.

1 Einleitung

Oft ist es notwendig, einen Datenfluss, d.h. einen Strom von Paketen von der Quelle zum Ziel, auf seine Dienstgüte (Quality-of-Service, QoS) hin zu untersuchen. Interessierende Kennzahlen sind dabei zum Beispiel Bandbreite, Jitter, Zuverlässigkeit, Antwortzeiten, und nicht zuletzt auch die Verzögerung. Unter (Einweg-) Verzögerung (oder *One-Way-Delay*) verstehen wir dabei die Zeit, die ein Datenpaket benötigt, um vollständig von einer Quelle bis zum Ziel zu gelangen. Die Verzögerung kann, abhängig von der verwendeten Hardware und der Auslastung des Netzwerks stark variieren. Aktive Netzwerkkomponenten wie Switches oder Router (im Gegensatz zu passiven Komponenten wie Repeater), aber auch Länge und Art des Übertragungsmediums haben einen grossen Einfluss darauf.

Es existieren hoch spezialisierte Geräte, die für die Bestimmung von One-Way-Delays herangezogen werden können. Die Messung ist dabei, je nach verwendetem Gerät, bis auf Bruchteile von Mikrosekunden genau. Leider ist ein solches Gerät mit einem nicht unerheblichen finanziellen Aufwand verbunden. Ein weiteres Problem tritt zu Tage, wenn man One-Way-Delay-Messungen zwischen zwei entfernten Endpunkten durchführen möchte. Diese Geräte sind i.d.R. nur für lokale Messungen ausgelegt – Quelle und Senke müssen sich also in unmittelbarer Nähe zueinander befinden. Eine Messung zwischen zwei Hosts, die sich in verschiedenen Städten oder sogar auf verschiedenen Kontinenten befinden, ist somit unmöglich. Der Einfachheit halber verwendet man für solche Fälle oft die Round-Trip-Time – wie sie z.B. `ping` liefert – als Mass für die Verzögerung. `ping` misst die Zeit, die ein Paket benötigt, um *von der Quelle zum Ziel und wieder zurück* zu gelangen. Was wir erhalten, ist also die Summe der Verzögerungszeiten von Hin- und Rückweg sowie der Verarbeitungszeit im entfernten Host. Die Zeitmessung erfolgt ausschliesslich lokal auf dem `ping`-Client. Daraus mit Division durch zwei Rückschlüsse auf die jeweiligen One-Way-Delays zu ziehen, führt aber offensichtlich zu Problemen: Der Kommunikationsweg in einem verbindungslosen Netz wie dem Internet ist in der Regel asymmetrisch, was bedeutet, dass ein Paket nicht zwangsläufig auf demselben Weg zum Client zurück findet, auf dem es zuvor verschickt worden ist. Hin- und Rückweg können also durchaus unterschiedliche One-Way-Delays aufweisen. Eine Messung mit `ping` trägt diesem Umstand nicht Rechnung.

Zur Bestimmung einer Einweg-Verzögerung führt kein Weg an exakt synchronisierten Uhren vorbei. Alle beteiligten Hosts müssen Zugriff auf ein und dieselbe *globale Zeitbasis* haben. In der Praxis heisst dies, dass alle Hosts über interne Uhren verfügen müssen, die bis auf wenige Mikrosekunden genau synchronisiert sind. Genau diese Synchronisation verursacht aber Probleme, denn eine akkurate Synchronisation von Computeruhren ist keine triviale Angelegenheit. Schon wenn man versucht, die Uhren zweier Hosts exakt zur selben Zeit auf Null zurückzustellen, stösst man an Grenzen. Es stellt sich die Frage, wie ein „Nullpunkt“ zwei Hosts exakt zur selben Zeit signalisiert werden kann. Und wie soll dies geschehen, wenn die Hosts dazu noch tausende Kilometer voneinander entfernt stehen? Selbst wenn das „Nullsetzen“ gelänge, würden die beiden Uhren bereits nach kurzer Zeit differieren („*Drift*“). Dies deshalb, weil die Oszillatoren von Computeruhren nie mit exakt derselben Frequenz schwingen, bzw. weil äussere Einflüsse wie z.B. Temteraturschwankungen die Frequenz der Oszillatoren beeinflussen.

Eine Möglichkeit, eine globale Zeitbasis herzustellen, ist NTP – ein Protokoll basierend auf UDP-Paketen, mit welchem sich Systemuhren über das Netzwerk synchronisieren lassen. NTP benutzt eine Hierarchie aus mehreren Schichten von Servern und Clients deren Wurzel ein ausgezeichneter Server bildet, der sog. Stratum-1-Server. Die nächst tiefere Schicht bezeichnet man als Stratum-2, usw. Der Stratum-1-Server ist in der Regel mit einem genauen Zeitgeber, einer Funkuhr o.ä. ausgestattet. Er dient als Referenz für alle tieferen Schichten. Ein NTP-Client holt sich durch Austausch von Paketen mit Zeit- und Offsetinformationen die Referenzzeit von einem oder mehreren Servern aus höherliegenden Schichten. Mit Hilfe von Kennzahlen wie z.B. Erreichbarkeit des Servers, Antwortzeiten (Delay) und Schwankung der Antwortzeiten (Jitter), wählt der Client dann von allen möglichen Zeitquellen die beste aus. Lokal *diszipliniert* der NTP-Daemon (`ntpd`), der auch für den Paketaustausch zuständig ist, die Systemuhr, indem er ihre Frequenz geringfügig anpasst oder sogar Schaltsekunden einfügt. Die paketbasierte Zeitsynchronisation mit NTP leidet aber unter zwei entscheidenden Nachteilen: Da die Pakete mit den Refernezzeiten selbst über das Netz geschickt werden müssen und somit unter Delay und Jitter leiden, wird eine Synchronisation nur auf einige Millisekunden oder bestenfalls auf einige hundert Mikrosekunden genau erreicht [9] – und dies auch nur, nachdem die Hosts einige Stunden lang Synchronisationspakete ausgetauscht haben. Ein weiterer, wenn auch nicht so gravierender Nachteil von NTP besteht darin, dass die Zeitsynchronisation über dasselbe Medium erfolgt, wie die eigentliche One-Way-Delay-Messung. Eine Verfälschung der Messungen kann also nicht ganz ausgeschlossen werden. Für Anwendungen wie One-Way-Delay-Messungen, die Mikrosekundengenauigkeit verlangen, ist NTP also nicht zu gebrauchen.

Eine weitere Möglichkeit, die Uhren zu synchronisieren, besteht darin, jeden Computer mit einer hochpräzisen (Atom-)Uhr auszustatten. Solche Instrumente sind jedoch mit erheblichen Kosten verbunden und somit nur in Ausnahmefällen praktikabel. Das satellitengestützte Navigationssystem GPS und der Anschluss ensprechender Empfangsgeräte an die Hosts bieten hier nun eine Alternative. GPS-Empfänger können dazu verwendet werden, die Zeit eines Hosts mit einer Genauigkeit von einigen Mikrosekunden zu setzen und somit für eine akzeptable globale Zeitbasis zu sorgen. Ausserdem sind diese Geräte mittlerweile relativ preiswert.

Das Ziel dieser Projektarbeit ist die Integration einer externen GPS-Uhr in Linux bzw. in den Linux-Kern. Desweiteren beinhaltet die Arbeit die Erstellung eines kleinen Demonstrators, welcher den One-Way-Delay zwischen zwei mit GPS-Empfängern ausgestatteten Hosts bestimmt.

Kapitel 2 widmet sich der verwendeten GPS-Hardware. Es wird gezeigt, welche Zeitinformationen dem Computer durch den GPS-Empfänger zur Verfügung stehen.

Kapitel 3 beleuchtet die präzise Zeitmessung mit einem Computer. Es werden dabei zwei Ansätze vorgestellt.

Kapitel 4 beschreibt die Implementation eines Demonstrators (`gping`). Wie der Name vermuten lässt, erfüllt er ähnliche Aufgaben wie das Unix-Tool `ping`. In einem zweiten Teil werden einige Messungen und Evaluationen durchgeführt.

2 GPS-Empfänger

Aufgabe des GPS, des „Global Positioning System“, ist eigentlich die satellitengestützte Ortung von Objekten auf der Erde oder in deren Nähe. Das Funktionsprinzip ist dabei relativ einfach: Jeder Satellit sendet in einem festgelegten Abstand ein Trägersignal, welches die Position des Satelliten sowie die exakte Sendezeit enthält. Empfängt ein Gerät am Boden solche Signale von drei verschiedenen Satelliten, ist eine Positionsbestimmung möglich. Die Anforderungen an die Zeitgenauigkeit sind dabei enorm. Ein Laufzeitfehler von nur einer Tausendstel Sekunde würde einen Distanzfehler von ca. 300 km bedeuten, wodurch natürlich das ganze System unbrauchbar würde. Die geforderte Präzision wird mit Atomuhren an Bord jedes einzelnen der 24 GPS-Satelliten erreicht. Quasi als Nebenprodukt der Positionsbestimmung liefert GPS auch ein genaues Zeitsignal, welches sich sehr gut als Referenz für Computeruhren eignet. Theoretisch lässt sich eine Genauigkeit von gegen 340 Nanosekunden erreichen [1].

In dieser Arbeit werden zwei GPS-Empfänger GPSClock 200 verwendet [2]. Sie sind speziell für Zeitapplikationen ausgelegt und versprechen laut Dokumentation eine Genauigkeit von unter einer Mikrosekunde (1µs). Die GPSClock 200 wird über eine RS232-Schnittstelle an den Computer angeschlossen. Das mitgelieferte Anschlusskabel muss vor dem Gebrauch noch mit einem D-SUB-9 Stecker versehen werden, was aber mit ein wenig

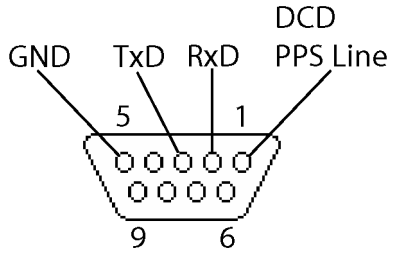

Abbildung 1: D-SUB-9-Stecker (female, front). Die GPSClock 200 verwendet das RS232-Protokoll.

Lötaufwand ohne weiteres zu bewerkstelligen ist. Nach dem Einschalten des GPS-Empfängers findet man die Meldungen des GPS-Empfängers auf dem RxD-Pin. In festen Intervallen meldet das Gerät durch sogenannte *Sentences* Positions-, Geschwindigkeits- und Zeitdaten. Die Sentences sind dabei einfache Textstrings, die gemäss dem NMEA-0183-Protokoll („National Marine Electronics Association“) geformt sind [2][10]. Ein Sentence beginnt immer mit einem Dollarzeichen, gefolgt von zwei Zeichen mit einer Gerätekennung sowie einem Befehl, bestehend aus drei Zeichen (s.u.). Eine Liste mit möglichen Sentences sowie eine kurze Beschreibung findet man u.a. im Usermanual der GPS-Clock. Für diese Arbeit ist vorallem ein Typ von Interesse: Der sog. ZDA-Sentence („Time & Date“) beinhaltet das vollständige Datum und die Zeit der nächsten vollen Sekunde.

```
# Beispiel eines ZDA-Sentences
# 20. Juni 1999, 7h 23m 17s

$GPZDA,072317,20,06,1999,+00,00<CR><LF>
```

Jeweils ca. 400 ms nach Eintreffen des ZDA-Sentences, lässt sich ein elektrischer Impuls an der PPS-Leitung messen (Abb. 2). Die ansteigende Flanke dieses Stromimpulses markiert dabei den Beginn der neuen Sekunde, synchron mit der Uhrzeit, welche der ZDA-Sentence zuvor mitgeteilt hat. Das Ganze funktioniert also ähnlich wie die Telefon-Zeitansage: Eine Stimme verrät die Uhrzeit, die dann beim nächsten Signal aktuell wird.

Die GPSClock 200 beginnt ca. 45 Sekunden nach dem Einschalten, im Sekundentakt diese sog. *PPS*-Signale („Pulse per Second“) auf einer speziellen Leitung („PPS-Line“) abzusetzen. Es kann vorkommen, z.B. nach einer grösseren Positionsveränderung, dass sich der GPS-Empfänger mit den Satelliten neu kalibrieren muss,

was ein Aussetzen des PPS-Signals für einige Minuten bis sogar Stunden zur Folge hat. Die PPS-Leitung wird normalerweise auf den DCD-Pin

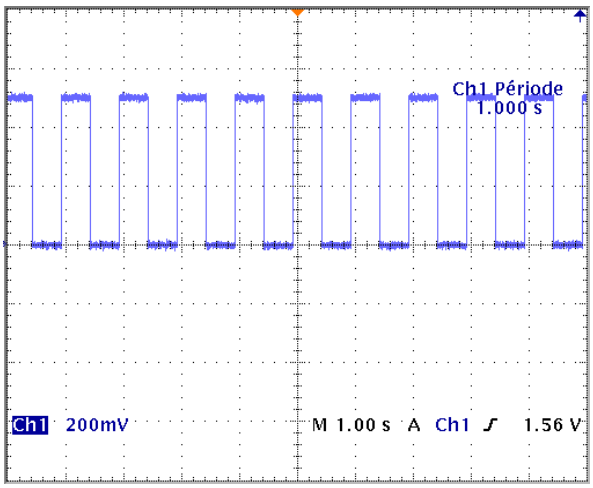

Abbildung 2: Das PPS-Signal gemessen mit einer 1:10 Sonde am DCD-Pin des D-SUB-9 Steckers. Die Periode beträgt genau 1 Sekunde, die Amplitude +5V. Der Pegel bleibt genau 0.5 Sekunden auf high.

(Abb. 1) des D-DUB-9-Steckers gelegt, wodurch bei Eintreffen eines PPS-Signals, und mit entsprechender Programmierung der Hardwarebausteine, ein Interrupt ausgelöst wird.

Folgende Aufstellung fasst die grundsätzlichen Schritte für den Gebrauch des GPS-Empfängers als genauen Zeitgeber kurz zusammen:

1. Lesen des ZDA-Sentences auf der seriellen Schnittstelle.
2. Parsen der gemeldeten Zeit.
3. Warten auf das PPS-Signal.
4. Nach Auftreten des Signals die kernelinternen Zeitvariablen anpassen.

Eine vollständige und ununterbrochene Abarbeitung dieser Liste ist dabei nicht unbedingt nötig. Oft reicht auch ein Intervall von z.B. 16 oder auch mehr Sekunden. Wie gross genau der Intervall sein sollte, ist von der verwendeten Hardware und vom internen Zeitmodell abhängig.

3 Präzise Zeitmessung mit dem Computer

Wie in der Einleitung bereits dargelegt, verlangt die Messung von One-Way-Delays zwischen zwei Endsystemen, dass deren Uhren exakt synchronisiert sind, d.h. ihre Abweichung untereinander muss sich in einem markant kleineren Rahmen bewegen, als die Zeiten, die effektiv gemessen werden sollen. Weiter muss die Uhr eine genügend grosse Auflösung bieten. Im Falle von One-Way-Delays ist dies eine Auflösung von mindestens Mikrosekunden. Das Standard-Zeitmodell von Linux bietet zwar eine solche Auflösung (µs), jedoch sind die Mittel, Uhr und GPS-Clock zu integrieren sehr begrenzt. Für eine solche Integration sind Änderungen in den Kernelsourcen oder mindestens zusätzliche Softwarekomponenten unerlässlich. In dieser Arbeit werden dazu nun zwei Ansätze verfolgt: a) eine eigene Implementation („PPSClock“) einer genauen Uhr mit PPS-Unterstützung in Form eines Kernelmoduls sowie eines Daemons, und b) eine bereits bestehende Implementation („PPSKit“), welche in die Kernelsourcen gepacht werden muss und danach mit dem NTP-Daemon (`ntpd`) zusammenarbeitet. Die Gründe, warum es Sinn macht, nebst dem bestehenden PPSKit auch die Implementation einer komplett neuen Lösung ins Auge zu fassen, sollen hier kurz zusammengefasst werden:

- Im Gegensatz zu einem Kernelpatch verspricht ein Modul eine höhere Flexibilität. Es muss nicht der ganze Kernel neu übersetzt werden, was dem User (oder besser: dem Administrator) bei der Installation einiges an Arbeit erspart.
- Der Kernelpatch existiert gut zwei Jahre nach Erscheinen von Kernel 2.6 immer noch ausschliesslich für die 2.4-Linie des Kerns. Die PPSClock wurde für Kernel 2.6 implementiert.
- Ein Einsatz von PPSkit in Verbindung mit `ntpd` benötigt einige Zeit (½ Stunde bis mehrere Stunden), bis eine annehmbare Synchronisation erreicht wird. Bei der PPS-Clock ist eine Messung schon nach 8 Sekunden möglich.

Leider ist aber festzustellen, dass eine Implementation einer Uhr als Modul nie dieselbe Genauigkeit erreichen kann, wie eine Implementation direkt im Kern. Der Grund dafür liegt darin, dass ein Modul keinen Zugriff auf den Timerinterrupt des Systems erlangen kann – es sei denn, es werden Techniken wie z.B. Interrupt-Hooking verwendet, welche aber alles andere als elegant und portabel sind.

Dieser Abschnitt beschreibt zu Beginn den Aufbau und die Funktionsweise einer neuen Implementation einer PPS-unterstützten Uhr, der *PPSClock*. Anschliessend wird der Fokus auf das PPSKit in Verbindung mit dem NTP-Daemon `ntpd` gelegt.

3.1 PPSClock

Zu allererst werden einige Richtlinien bzw. Zielvorgaben formuliert, welche einen Rahmen für das fertige Produkt vorgeben sollen. Sie sollen eine gewisse Hilfestellung bei konkreten Fragen betreffend Softwarearchitektur und Implementation bieten. Die PPSClock:

- soll einfach installierbar sein.

- velangt keine Änderungen im Quellcode des Linux-Kerns. Ein Patchen des Kerns mit anschliessender Neuübersetzung ist mit einem grossen Aufwand für den User verbunden.
- führt so wenig Code wie möglich im Kernelspace aus.
- verzichtet auf grosse und unübersichtliche Programmpakete wie z.B. NTP.
- erreicht eine Genauigkeit, mit welcher sich One-Way-Delays auf kurzen Netzwerkabschnitten ermitteln lassen.
- ist bereits nach kurzer Zeit für Messungen bereit.

Mit Blick auf obige Vorgaben, wurde eine Softwarelösung erarbeitet, die im folgenden besprochen werden soll. Die Abschnitte 3.1.2 bis 3.1.4 erlauben anschliessend einen etwas tieferen Einblick in die Interna der einzelnen Komponenten und gehen auf spezifische Fragen der Implementation ein.

3.1.1 Design und Ansatz

Die softwareseitige Intergration des GPS-Gerätes in Linux erfolgt in zwei Teilen und ist in Abbildung 3 schematisch dargestellt:

1. `ppsdrv.ko`: Ein Kernelmodul, welches sich u.a. mit der Verarbeitung der PPS-Signale bzw. der dadurch ausgelösten Interrupts beschäftigt. Zusätzlich stellt es ein Character Device zur Verfügung, welches Programme abfragen können, um die aktuelle Zeit in Erfahrung zu bringen.
2. `ppsd`: Der Userspace Daemon liest und verarbeitet die GPS-Sentences. Er verwendet die vom Modul bereitgestellten Ioctls, um wenn nötig die PPS-Clock anzupassen bzw. zu setzen.

Grundsätzlich beginnt der Lebenszyklus der PPS-Clock mit dem Laden des Kernelmoduls `ppsdrv.ko`. Es folgt eine Initialisierungsphase, die 8 Sekunden lang dauert. Nach der Initialisierungsphase ist die Clock für die Zeitmessung bereit. Ein auftretendes PPS-Signal veranlasst das Modul, die PPS-Clock um eine Sekunde hochzuzählen. Der `ppsd`-Daemon wird gestartet, nachdem das Modul geladen wurde. Er liest in bestimmten Abständen, z.B. 60 Sekunden, die Zeitmeldungen (Sentences) von der seriellen Schnittstelle und setzt die PPS-Clock entsprechend. Dies muss periodisch geschehen, da es unter Umständen möglich ist, dass das PPS-Signal kurzfristig aussetzt und somit eine Sekunde quasi „verloren“ geht. Das Modul stellt weiter durch ein Character Device eine Möglichkeit zur Verfügung, die aktuelle Zeit vom Userspace aus abzufragen.

3.1.2 Das Zeitmodell

Zur Zeitberechnung wird das Cycle-Count-Register („PCC“), welches auf fast allen modernen CPUs vohanden ist, herangezogen. Auf heutigen Systemen mit 1 GHz und mehr ist dies der Zähler, mit der höchsten Auflösung, weshalb er sich für Messungen im Mikrosekundenbereich relativ gut eignet. Zu beachten ist hier, dass es gelegentlich

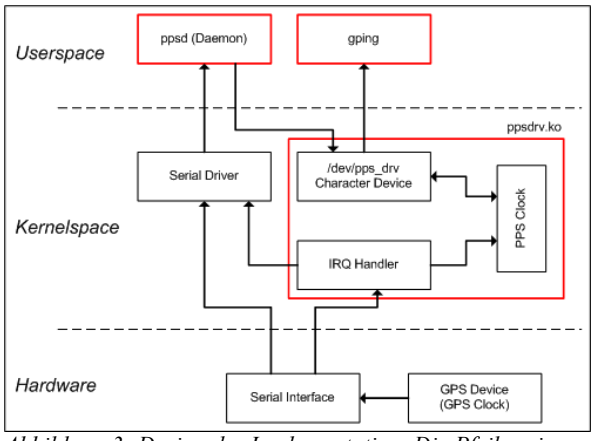

Abbildung 3: Design der Implementation. Die Pfeile zeigen die Richtung des Datenflusses an. Rot: speziell für diese Arbeit geschriebene Softwarekomponenten.

Systeme gibt, die bei wenig Aktivität die CPU-Frequenz hinnterschalten. Dadurch würde die Zeitmessung mit der hier vorgeschlagenen Methode bis zur Unbrauchbarkeit verfälscht und die PPS-Clock muss in eine neue Initialisierungsphase eintreten. Bei der PPSClock geschieht dies ohne fremdes Zutun. Nach jeweils 8 Sekunden hat sich die PPSClock aber bereits der neuen Situation angepasst. Das grosse Problem, welchem man sich beim Cycle-Count-Register aber gegenüber sieht ist nun, dass es sich nicht eigentlich um einen *Zeit*zähler handelt – viel mehr ist es einfach ein Zähler, der jeden CPU-Schritt mitzählt und vordergründig nichts mit einer Uhrzeit o.ä. zu tun hat. Irgendwie muss man also die aktuelle Uhrzeit auf diese „CPU-interne Zeit“ abbilden. Folgendes soll den Vorgang des Abbildens verdeutlichen.

Als erstes wird eine monoton wachsende Zeitvariable $vTime_{t0}$ definiert. Der in ihr enthaltene Wert repräsentiert einen eindeutigen Punkt auf der Zeitachse, nämlich den Beginn einer Sekunde. Zu Beginn wird sie mit dem aktuellen Datum und der Uhrzeit initialisiert. Gleichzeitig wird der aktuelle Wert des CPU-spezifischen Cycle-Count-Registers gespeichert („TSC-Register“ auf Intel). Die Auflösung von $vTime_{t0}$ beträgt eine Sekunde. Mit jeder verstrichenen Sekunde – gekennzeichnet durch das PPS-Signal – wird sie um Eins inkrementiert und der zugehörige neue TSC-Wert in TSC_{t0} gespeichert. $vTime_{t0}$ und TSC_{t0} ermöglichen zusammen also ein Mapping zwischen der CPU-internen Zeit (TSC-Registerwert) und der realen Zeit. Eine weitere Variable TSC_{delta} zählt, wieviele CPU-Zyklen innerhalb einer Sekunde

verstreichen; sie dient also als genaue CPU-Frequenzangabe in Hz. Kommt nun z.B. über eine spezielle Funktion eine Zeitanfrage, so kann die aktuelle Zeit mit Hilfe der erwähnten Variablen interpoliert werden:

$$vTime_{current} = vTime_{t0} + \frac{TSC_{current} - TSC_{t0}}{TSC_{delta}}$$

Die Interpolation kann entweder im Kernel- oder, durch Übergabe der Daten z.B. mittels Ioctl, im Userspace erfolgen. Zu beachten ist dabei, dass bei den meisten Betriebssystemen, so auch bei Linux, im Kernelmode keine Fliesskommaberechnungen und keine Divisionen von long-long-Datentypen unterstützt werden. Will man die Interpolation trotzdem im Kernelspace durchführen, müssen spezielle Vorkehrungen getroffen werden, um eventuelle Überläufe o.Ä. abzufangen. Trotz dieses Mehraufwandes ist die Berechnung im Kernelspace derjenigen im Userspace vorzuziehen. Denn würde sie im Userspace durchgeführt, so könnte der Scheduler des Betriebsystems die Berechnung evtl. unterbrechen und erst zu einem späteren Zeitpunkt weiterführen. Ein solches Verhalten verfälscht jedes Resultat und ist für eine genaue Zeitmessung inakzeptabel.

3.1.3 Das Kernelmodul – ppsdrv.ko

Das Kernelmodul `ppsdrv.ko` ist – einfach ausgedrückt – eine neue Kernel-Uhr. Im Kernelmodul befinden sich die oben erwähnten Variablen $vTime_{t0}$, TSC_{t0} sowie TSC_{delta}. Mit Hilfe dieser Variablen lässt sich die aktuelle Uhrzeit berechnen. Weiter stellt das Modul auch den IRQ-Handler zur Behandlung der PPS-Signale sowie ein Character Device für die Kommunikation mit dem User zur Verfügung.

IRQ-Handler

Zentral für eine in Software implementierte Uhr ist ein Mechanismus, welcher sie über eine bestimmte abgelaufene Zeitspanne informiert und die Uhr so ihre internen Variablen der neuen „aktuellen“ Zeit anpassen kann. Als Beispiel sei die traditionelle Systemuhr in Linux genannt. Sie bemüht den Timerinterrupt, der in Kernel 2.6 1000 mal pro Sekunde ausgelöst wird. Ein Auftreten eines Timerinterrupts veranlasst den Linuxkern, die Sytemuhr um 1/1000 Sekunde hochzuzählen. Die PPS-Clock funktioniert im Grunde ähnlich. Sie verwendet aber statt des Timerinterrupts das PPS-Signal, welches durch die serielle Schnittstelle einen Interrupt auslöst. Um auf diesen Interrupt reagieren zu können, ist es nötig, sich in das Interrupthandling des Kerns einzuklinken. Mit Hilfe der Kernelfunktion `request_irq()` ([4], Kapitel 10) wird ein eigener Interrupthandler für die serielle Schnittstelle installiert. Hier ist zu beachten, dass bei der Übersetzung des Kernels der Treiber für die serielle Schnittstelle unbedingt als Modul und mit der Option für Interruptsharing kompiliert werden muss, ansonsten schlägt ein Aufruf von `request_irq()` fehl. Jedesmal wenn die serielle Schnittstelle also einen Interrupt auslöst – was grob gesagt bei jedem Zustandswechsel der Steuerleitungen der Fall ist – wird der neue IRQ-Handler ausgeführt. Er merkt sich den aktuellen Wert des TSC-Registers und stellt danach fest, ob der gerade gefeuerte Interrupt einer ist, der uns interessiert, d.h. ob der Interrupt durch ein PPS-Signal ausgelöst wurde. Ist dies nicht der Fall, wird die Abarbeitung des Handlers ohne weitere Verzögerung abgebrochen. Wird jedoch ein PPS-Signal erkannt, so kann der Handler seine Timekeeping-Aufgaben wahrnehmen und die interne Uhr um eine Sekunde voran schreiten lassen: Der zuvor gespeicherte TSC-Wert wird zu TSC_{t0}, $vTime_{t0}$ um eine Sekunde erhöht und TSC_{delta} aktualisiert. Eine neue Zeitberechnung würde nun nach dem Schema von 3.1.2 mit diesen neuen Basisdaten erfolgen.

Die Verarbeitung des PPS-Signals sollte natürlich möglichst ohne Verzögerung veranlasst werden. Leider kann aber mit der vorgeschlagenen Methode (`request_irq()`) eine gewisse Verzögerung weder ganz ausgeschaltet, noch genau vorhergesagt werden. Die Gründe dafür sind zweierlei: Erstens befindet sich der neu installierte IRQ-Handler in einer Schlange von mehreren Handlern, die vom Betriebssystem verwaltet wird. Der Ausführungszeitpunkt hängt also von der Güte und Effizienz von Linux ab. Zweitens gehört die serielle Schnittstelle, oder viel mehr ihr Interrupt, faktisch zu den Geräten, mit der niedrigsten Priorität im System. Die meisten Interrupts können also den der seriellen Schnittstelle aufgrund seiner niedrigen Priorität verzögern. Dazu gehören z.B. der Timerinterrupt oder auch die Interrupts des IDE-Controllers. Ersteres kann zum Beispiel mit „Interrupt-Hooking“ umgangen werden. Die Folge ist aber sehr unschöner und vorallem hochgradig plattformabhängiger Code, weshalb der Weg über `request_irq()` dem Interrupt-Hooking vorzuziehen ist. Um dem zweiten Problem entgegenzuwirken, könnte man versuchen, die Interruptprioritäten des Systems anzupassen. Auf neueren PC-Systemen mit APIC („Advanced Programmable Interrupt Controller“) ist dies bis zu einem gewissen Grad möglich. Bei Tests erwiesen sich die genannten Verzögerungen aber als nicht so gravierend, als dass sich weitere grosse Zeitinvestitionen in diesem Feld lohnen würden.

Kommunikation

Die Kommunikation des Moduls mit der Aussenwelt, sprich, mit dem Userspace, erfolgt über ein Character Device ([4], Kapitel 3). Es wird ebenfalls vom Kernelmodul zur Verfügung gestellt.

Interessiert sich ein Programm für die aktuelle Uhrzeit, so führt es eine Read-Operation auf dem Device aus. Diese Read-Operation veranlasst Linux, in den Kernelmodus zu wechseln und die Kontrolle dem Kernelmodul zu übergeben, welches dann seinerseits die aktuelle Uhrzeit interpoliert und an den User-Prozess zurückgibt. Die zurückgelieferten Daten sind in einer `timespec`-Struktur verpackt, was eine weitere Verarbeitung mit den libc-Bibliotheksfunktionen möglich macht. Weil immer eine `timespec`-Struktur zurückgeliefert werden soll, werden nur Leseoperationen von `sizeof(struct timespec)` Bytes unterstützt. Alle anderen Anfragen werden mit einem `EINVAL` quittiert.

Auch das Setzen der PPS-Clock erfolgt über das Character Device. Hier werden allerdings Ioctls und nicht die normalen Read-/Write-Operationen verwendet. Mit einem Aufruf von `ioctl()` im Userspace mit der Option `PPS_SET_TIME` sowie einer `timespec`-Struktur als Argument kann die PPS-Clock neu gesetzt werden. Die neue Uhrzeit wird erst beim nächsten PPS-Signal aktiv.

Modulparameter

Das Modul `ppsdrv.ko` unterstützt zum Ladezeitpunkt einen Parameter, `use_port`. `use_port` teilt dem Modul mit, welcher serielle Port verwendet wird.

```
# insmod ppsdrv.ko use_port=0
```

bewirkt, dass die erste serielle Schnittstelle verwendet werden soll; bei `use_port=1` kommt die zweite zum Zug usw. Wird der Parameter weggelassen, wird `use_port=0` angenommen.

3.1.4 Userspace Daemon – ppsd

Während sich die PPS-Clock resp. das Kernelmodul ausschliesslich um die PPS-Signale kümmert, besteht die Aufgabe des Daemons in der Verarbeitung der Sentences, die die GPS-Clock liefert. Grundsätzlich könnte man beide Aufgaben auch in ein einziges Kernelmodul packen. Der Vorteil wäre ein einfacheres Aufsetzen des gesamten Systems, weil ja nur noch eine Komponente für das korrekte Funktionieren installiert werden müsste. Für die PPSClock wurde aber dennoch ein eigener Daemon erstellt. Die Gründe dafür sind relativ einfach einzusehen: Das Lesen von der seriellen Schnittstelle sowie das Parsing der GPS-Sentences sind keine ausserordentlich zeitkritischen Operationen. Nach dem Eintreffen des ZDA-Sentences hat das System ganze 300 bis 400 ms Zeit, die Informationen zu parsen und weiterzureichen. Dies ist mehr als genug für eine solch triviale Aufgabe. Eine Einbettung in den Kern ist also nicht nötig und wäre ausserdem gefährlich. Es würde ohne nennenswerten zusätzlichen Nutzen die Anfälligkeit für Fehler und somit für einen Systemabsturz markant erhöhen.

Funktionsweise

Der Daemon liest in bestimmten, zur Compilezeit festgelegten Abständen den ZDA-Sentence auf der seriellen Schnittstelle. Aus diesem Sentence wird das Datum und die Uhrzeit extrahiert und in eine `timeval`-Struktur verpackt. Durch einen Ioctl-Aufruf auf das Character Device `/dev/pps_drv` wird der PPSClock die neue Zeit mitgeteilt. Beim nächsten PPS-Signal vollzieht das Modul einen Wechsel zur neuen Zeitbasis und arbeitet von diesem Zeitpunkt an mit der neuen Zeit weiter. Der Daemon hat nach der Übergabe der `timeval`-Struktur an das Modul seine Schuldigkeit vorerst getan und legt sich für eine Zeit von `ZDA_READ_INTERVAL` Sekunden schlafen. Nach Ablauf dieser Zeitspanne beginnt das ganze von vorn. Während der Daemon schläft, läuft das Modul ganz normal weiter und wartet auf PPS-Signale.

3.2 PPSKit in Verbindung mit ntpd

Der zweite Ansatz, der hier für eine präzise Zeitmessung verfolgt wird, benötigt einen Kernelpatch sowie den NTP-Daemon. Der Patch kann als *PPSKit*, wie auch die Kernelsourcen selber, von [6] runtergeladen werden. Leider existiert das Kit, auch gute zwei Jahre nach Erscheinen von Kernel 2.6, nur für Kernel 2.4. Es gibt zwar eine Alpha-Version für Kernel 2.6, welche aber noch nicht wirklich stabil läuft. Aus diesem Grund verwenden wir für diesen zweiten Ansatz ausschliesslich den Kernel 2.4.29 mit dem enstprechenden Patch. Die zweite notwendige Komponente, der NTP-Daemon `ntpd`, kommt als Teil des NTP-Pakets [9].

3.2.1 PPSkit

Im Gegensatz zur vorgängig vorgestellten PPSClock, führt das PPSkit keine komplett neue Uhr im System ein. Die existierende API für die Systemuhr wird grundsätzlich beibehalten. Das PPSKit ist im Wesentlichen eine Implementation der RFCs 2783 [3] und 1589 [5]. Für eine ausführlichere Beschreibung sei auf die entsprechenden Dokumente verwiesen.

Uhr mit Nanosekundenauflösung

Der Patch erweitert die Kerneluhr („Systemuhr“) von einer Auflösung von Mikrosekunden zu einer Uhr mit Nanosekundenauflösung. Das Prinzip dieser Uhr bleibt hier dasselbe wie bei der PPSClock und auch bei der Standard-Systemuhr: Die Uhr wird jeweils nach Ablauf einer festen Zeitspanne aktualisiert. Der Zeitpunkt der Akualisierung wird *Tick* genannt. Ein Zeitpunkt zwischen zwei Ticks wird mit Hilfe des Cycle-Count-Registers interpoliert, wenn eine entsprechende Anfrage vorliegt.

Neue Systemcalls

Unixprogramme können die aktuelle Uhrzeit sowie Informationen über Zeitzonen mit dem `gettimeofday`-Systemcall abfragen. Für bestimmte Applikationen wäre es aber wünschenswert, z.B. den maximalen Fehler der gemeldeten Zeit, Frequenzschwankungen des CPU-Oszillators und noch weitere Kennzahlen ermitteln zu können. RFC 1589 schlägt deshalb zwei neue Systemcalls vor: `ntp_gettime()` und `ntp_adjtime()`. `ntp_gettime()` liefert neben der aktuellen Zeit auch noch den maximalen Fehler und den geschätzten Fehler einer Zeitmessung. `ntp_adjtime()` kann verwendet werden, um andere kernelinterne Zeitvariablen zu lesen oder zu schreiben. Dazu gehören z.B. Variablen betr. Frequenzanpassungen des CPU-Oszillators, Warnungen für Schaltsekunden oder Ähnliches. Das PPS-Kit fügt diese neuen Systemcalls in den Kernel ein. Der NTP-Daemon kann sie verwenden und erhält so zusätzliche Informationen bzw. Mittel, welche er in den Synchronisationsprozess einfliessen lässt.

PPS-API

RFC 2783 schlägt einen standardisierten Weg vor, wie mit PPS-Quellen umgegangen werden soll. Das Dokument beschreibt eine neue API („PPS-API“) für Userspace-Applikationen, mit welcher PPS-Quellen verwaltet und abgefragt werden können. Das PPSKit stellt die PPS-API in Form einer include-Datei (`timepps.h`) mit entsprechenden Makros bzw. inline-Funktionen zur Verfügung. Mit

```
rv = time_pps_create(filedescriptor,
       &handle);
```

kann z.B. ein Gerät als neue PPS-Quelle registriert werden. Hinter den Kulissen werden durch den Aufruf von Funktionen der PPS-API bestimmte Ioctls auf das Gerät `filedescriptor` ausgeübt. Natürlich muss das Gerät die entsprechenden Ioctls auch unterstützen, ansonsten kann es nicht als PPS-Quelle benutzt werden. Der Rückgabewert `rv` erhält in einem solchen Fall einen negativen Wert. Mit dem PPSKit wird als Beispiel der Treiber für die serielle Schnittstelle unter anderem um genau diese Ioctls erweitert, so dass sie als PPS-Quelle dienen kann.

3.2.2 ntpd und PPSKit

In diesem Abschnitt soll das Zusammenspiel von NTP-Daemon und PPSKit erläutert werden. Es kommt der `ntpd` in der Version 4.2.0 zum Einsatz [9]. Jede frühere Version müsste für die Zusammenarbeit mit der GPSClock 200 zuerst gepacht werden. Der NTP-Daemon kann auf Linux mit oder auch ohne ein installiertes PPSKit betrieben werden. Die Entscheidung darüber wird bereits zur Compilezeit von NTP getroffen. Der Build-Prozess des Pakets stellt fest, ob die APIs, welche von [3] und [5] beschrieben werden, vorhanden sind. Wenn ja, werden sie automatisch miteinbezogen.

Die Aufgabe des NTP-Daemon `ntpd` besteht im Grunde darin, die lokale Uhr (Systemuhr) an eine Referenzuhr anzupassen und eventuell (im Servermodus) für andere Mitglieder des NTP-Netzes selber als Referenzuhr zu agieren. Letzteres ist für diese Arbeit irrlelevat, weshalb hier auch nicht speziell darauf eingegangen wird.

Eine Uhr, die für den `ntpd` als Referenz agiert, wird *Peer* genannt. Bei der Konfiguration des Daemons können mehrere Peers angegeben werden. Die Peers können dabei andere NTP-Server im Netzwerk, oder aber auch direkt an das System angeschlossene Geräte (z.B. spezielle

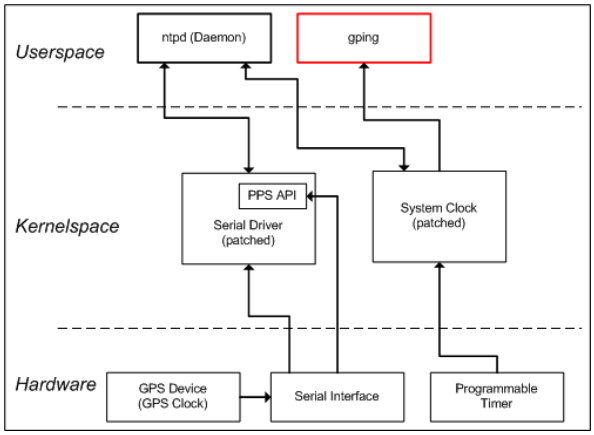

Abbildung 4: Der NTP-Daemon ntpd im Zusammenspiel mit dem gepatchten Kernel.

Atomuhren oder GPS-Empfänger) sein. Der Daemon holt sich periodisch jeweils die Referenzzeit seiner Peers und berechnet danach in einem relativ komplizierten Verfahren und unter Berücksichtigung verschiedener Kennzahlen (wie z.B. Erreichbarkeit des Peers, Jitter der eintreffenden Referenzzeiten, o.ä.) *die Referenzzeit*, nach der die lokale Uhr gerichtet werden soll. Zusätzlich zur Referenzzeit, zieht der Daemon mit Hilfe der neuen PPS-API allfällig vorhandene PPS-Quellen zur exakten Bestimmung des Sekundenbeginns bei. Nachdem also die Referenzzeit bekannt ist, muss die lokale Uhr nun entsprechend angepasst werden; die Uhr wird sog. *diszipliniert*. Das Disziplinieren erfolgt im Wesentlichen in zwei Schritten: 1. Feststellen, wie gross die Abweichung zwischen lokaler Uhr und Referenz ist, d.h. es wird ein Offsetwert berechnet, und 2. die Phase und Frequenz der Uhr so verändern, dass sie sich nach und nach der Referenz annähert. `ntpd` verwendet dazu die mit dem PPSKit neu eingeführten Systemcalls `ntp_adjtime()` und `ntp_gettime()` [5]. Abbildung 4 zeigt schematisch die Architektur eines Systems mit PPSKit, `ntpd` und einer lokalen GPS-Zeitquelle.

3.2.3 ntpd und One-Way-Delay-Measurements

NTP, oder besser gesagt, der NTP-Daemon `ntpd`, bietet die Möglichkeit, die lokale Uhr mit einem angeschlossenen externen Gerät wie z.B. einer GPS-Uhr zu synchronisieren. Dieser Betriebsmodus wird von NTP Stratum-1-Servern verwendet – Server auf höchster Ebene in einem NTP-Netz. Wenn wir also zwei exakt synchronisierte Hosts für One-Way-Delay-Measurements haben wollen, so liegt es nahe, zwei identische Stratum-1-Server aufzusetzen: Jeder Server verfügt über seine eigene GPS-Clock und `ntpd` läuft jeweils ausschliesslich lokal, d.h. ohne Kontakt zwecks Synchronisation zum jeweils anderen Server.

Das Aufsetzen eines solchen Servers ist relativ aufwändig. Es muss das NTP-Paket installiert und konfiguriert, der Kernel gepacht, konfiguriert, neu übersetzt und installiert sowie das System neu gestartet werden. Um das ganze ein wenig flexibler zu gestalten, kann z.B. speziell eine Linux-Live-CD angepasst und für den Betrieb von `ntpd` in Verbindung mit der GPSClock 200 vorbereitet werden. Anhang C gibt einen Überblick über die notwendigen Schritte für ein Remastering einer Knoppix-CD.

3.3 PPSkit vs. PPSClock

Um dieses Kapitel abzuschliessen, macht es Sinn, die zwei Ansätze noch kurz einander gegeüberstellen. Es ist im Moment leider nicht möglich, die beiden Uhren direkt miteinander zu vergleichen, d.h. es ist nicht möglich festzustellen, ob die Uhren auf ein und demselben System auch wirklich exakt dieselbe Zeit anzeigen. Dies liegt daran, dass PPSKit und PPSClock für unterschiedliche Kernelversionen implementiert sind. Es ist aber durchaus möglich, sich auch ohne direkten Vergleich ein grobes Bild über das Verhalten der beiden Uhren zu machen. Folgendes ist eine (natürlich unvollständige) Aufstellung der Vor- und Nachteile beider Ansätze:

- Ein Kernel mit PPSKit und `ntpd` benötigt einige Zeit (min. ½ Stunde), bis die Synchronisation ausreicht, um One-Way-Delay-Measurements durchzuführen. Die PPSClock ist hingegen schon nach 8 Sekunden bereit.
- Die PPSClock kann dynamisch geladen und auch wieder entfernt werden. Ein Neustart des Systems ist nicht erforderlich.
- Das PPSKit behält die bestehende API bei, d.h. Programme können ohne Veräderungen über die gewohnten Funktionen wie z.B. `gettimeofday()` auf die genaue Uhr zugreifen. PPSClock führt eine neue API ein, resp. stellt ein spezielles Character-Device zur Interaktion mit der Uhr zur Verfügung.
- Bei der PPSClock kann es gelegentlich vorkommen, dass die Uhr nach hinten gestellt wird. Das PPSKit verfolgt einen etwas sanfteren Weg und nimmt Einfluss auf die Frequenz der Uhr, wodurch Sprünge vermieden werden.
- Ein System mit PPSKit ist unter idealen Bedingungen auf ca. 5 Mikrosekunden genau. Die Genauigkeit der PPSClock variiert dagegen sehr stark auf unterschiedlichen Systemen.

Der letzte Punkt ist auch gleich der entscheidende Nachteil der PPSClock: Sie ist vollständig abhängig vom CPU-Oszillator. Wenn dieser nicht mit einer konstanten Frequenz tickt, so verschlechtert sich die Zeitmessung erheblich. Bei vielen für diese Arbeit untersuchten modernen Pentium-4-Systemen konnten nur sehr geringe Frequenzschwankungen festgestellt werden, so dass mit der PPSClock durchaus eine mit dem PPSKit vergleichbare Genauigkeit erreicht werden könnte. Es gibt aber auch Systeme (speziell ein Dell Laptop Latitude C840), welche sehr grosse Schwankungen aufweisen. Beim erwähnten Dell-Modell würde nur eine Genauikeit von gegen 40 Mikrosekunden erreicht. Dies ist zwar erheblich besser als eine Synchronisation z.B. mit NTP übers Netz. Es ist aber auch schon fast an der Grenze, um noch einigermassen verlässliche One-Way-Delay-Werte zu messen.

Alles in allem muss festgestellt werden, dass die PPSClock viel anfälliger auf äussere Einflüsse ist, als ein Kernel mit PPSKit.

4 Evaluation

Es soll gezeigt werden, dass es möglich ist, One-Way-Delay-Messungen mit normalen PC-Systemen durchzuführen, die mit GPS-Geräten ausgestattet sind. Zu Beginn muss hier aber gleich festgehalten werden, dass keiner der beiden unter Kapitel 3 vorgestellten Ansätze zur Zeitmessung bzw. Zeitsynchronisation restlos überzeugend ist: Die PPSClock ist ohne Frage zuwenig ausgereift und dementsprechend weit von einem Einsatz in einem produktiven Umfeld entfernt. Demgegenüber steht das PPSKit seit gut zehn Jahren in einem ständigen Entwicklungsprozess und ist relativ zuverlässig. Obwohl ein System mit PPSKit gegenüber der PPSClock einige Vorteile aufweist, sind aber auch dort Nachteile festzustellen. Die zwei grössten sind einerseits die fehlende Unterstützung für neue Kernelversionen, und andererseits die relativ lange Synchronisationsphase, was einen flexiblen Einsatz praktisch verunmöglicht.

Im Folgenden sollen einige Resultate von Messungen besprochen werden. Zu Beginn werden die notwendigen Vorbereitungen im

Versuchsaufbau erläutert sowie die verwendete Software für die Messungen kurz vorgestellt. Es folgen One-Way-Delay-Messungen in einem LAN auf einer kurzen geographischen Distanz, anschliessend soll noch der Einsatz über eine längere Distanz beleuchtet werden.

4.1 Vorbereitungen

4.1.1 PPSKit mit ntpd

Bevor der `ntpd` gestartet wird, sollte die Systemuhr in die Nähe der „richtigen" Zeit gebracht werden. Damit wird eine Synchronisation erheblich beschleunigt. Es reicht, einen der zahlreichen öffentlichen NTP-Server abzufragen. Z.B.

```
# ntpdate 130.60.75.72
```

Mit `ntpdate` wird die Systemuhr ohne Verzögerung oder Frequenzanpassungen derjenigen des Servers angepasst. Natürlich wird aufgrund des Jitters im Netzwerk nur eine bescheidene Synchronisation im Bereich von Millisekunden erreicht. Anschliessend kann der NTP-Daemon gestartet werden. Bei richtiger Konfiguration (Anhang B) übernimmt er die kontinuierliche Abfrage des GPS-Empfangsgerätes. Der Parameter `-d` versetzt ihn in den Debug-Modus, wodurch eine Fehlersuche erleichtert wird. Er kann natürlich auch weggelassen werden:

```
# ntpd -d
```

4.1.2 PPSClock

Zur Verwendung der PPSClock muss zuerst das `ppsdrv.ko`-Modul geladen, sowie das entsprechende Device-File erstellt werden. Diese Aufgaben kann ein Script übernehmen:

```
# ./loadppsdrv.sh
```

Anschliessend muss der Userspace-Daemon zur Ausführung gebracht werden. Der Parameter `-f` ist optional und bewirkt, dass der Daemon nicht in den Hintergrund geschickt wird.

```
# ./ppsd -f
```

Bereits nach kurzer Zeit ist die PPSClock für Zeitanfragen bereit.

4.2 Ersatz für ping: gping

One-Way-Delay-Measurements sind Messungen, zu denen ein einfacher PC mit einem Betriebsystem ohne spezielle Unterstützung für Zeitsynchronisation nicht in der Lage ist. In der Regel gehört ein Tool zur Bestimmung der Einweg-Verzögerung deshalb auch nicht zum Lieferumfang. Als Teil dieser Projektarbeit wurde also ein kleiner Demonstrator geschrieben, welcher eine solche Messung erlaubt: `gping`. Wie der Name vermuten lässt, erfüllt `gping` eine ähnliche Aufgabe wie das Standard-Tool `ping`. Es misst aber statt der Round-Trip-Time die One-Way-Delays von Hin- und Rückweg getrennt.

4.2.1 Funktionsweise

Es gibt einen grundlegenden Unterschied zu `ping`: `gping` benutzt das UDP-Protokoll, nicht ICMP. Dies ist auch der Grund, warum der Zielhost einen eigenen `gping`-Server laufen lassen muss. Bei ICMP übernimmt der im Betriebssystem integrierte IP-Stack die Aufgabe des Servers.

4.2.2 Paketformat

Das Paketformat von gping ist denkbar einfach: Es werden nur `timespec`-Strukturen ausgetauscht.

Der Client füllt zuerst eine `timespec`-Struktur *A* mit seiner lokalen Zeit, die er von der zuvor synchronisierten Uhr erfahren hat. Er schickt das ganze zum Server. Der wiederrum, merkt sich die Zeit bei Eintreffen eines solchen Pakets in einer eigenen Variable *B*. Anschliessend schickt er beide, *A* und *B*, zurück an den Client, der jetzt ein zweites mal seine Zeit nimmt (*C*). Der Client besitzt jetzt also ein Paket bestehend aus drei `timespec`-Strukturen. Die Differenzen *B-A* und *C-B* ergeben die One-Way-Delays für den Hin- und den Rückweg.

4.3 Messung innerhalb eines LAN

Die Messung von One-Way-Delays in einem LAN erlaubt es, äussere Einflüsse zu reduzieren resp. besser zu kontrollieren. Z.B. kann besser bestimmt werden, wie hoch die aktuelle Netzwerklast auf einzelnen Abschnitten liegt, oder welchen Weg das Datenpaket durch das LAN nimmt. Das Ziel ist hier, möglichst unverfälschte und nachvollziehbare Messresultate zu erhalten. Die Messung in einem LAN hat fast zwangsläufig zur Folge, dass die beteiligten Hosts geographisch nahe beieinander stehen. Es wurden insgesamt drei Messreihen erstellt. Die erste gilt als aussagekräftige Refenenzmessung und entstand mit Hilfe von Smartbits. Dies ist ein Industriestandard-Messinstrument, welches die One-Way-Delay-Messung in einem LAN auf Bruchteile von Mikrosekunden genau erlaubt. Die zweite Messreihe entstand auf Hosts, die unter dem PPSKit synchronisiert werden. Bei der dritten kam die PPSClock zum Einsatz.

4.3.1 Versuchsaufbau

Der Versuchsaufbau besteht aus insgesamt 4

Hosts und einem 10Mbit Switch, die in Serie geschaltet werden. Die beiden Systeme in der Mitte dienen als Router für Netzwerkverkehr (Abb. 5, R_1 bzw. R_2), der zwischen den äusseren beiden ausgetauscht wird. Die Messungen erfolgen mit dem vorhin vorgestellten Programm `gping`. Neben der One-Way-Delay-Messung läuft keine weitere Kommunikation über das Netzwerk.

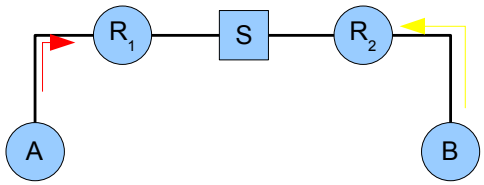

Abbildung 5: Versuchsaufbau für lokale Messungen. S: 10Mbit Switch; $R_{1,2}$: Router; A,B: Endsysteme mit PPSKit oder PPSClock.

4.3.2 Resultate

Abbildung 6 zeigt die Referenzmessung der One-Way-Delays im erwähnten Versuchsaufbau. Die Delays bewegen sich bei ungefähr 140µs, Hin- und Rückweg sind in etwa gleich und verlaufen

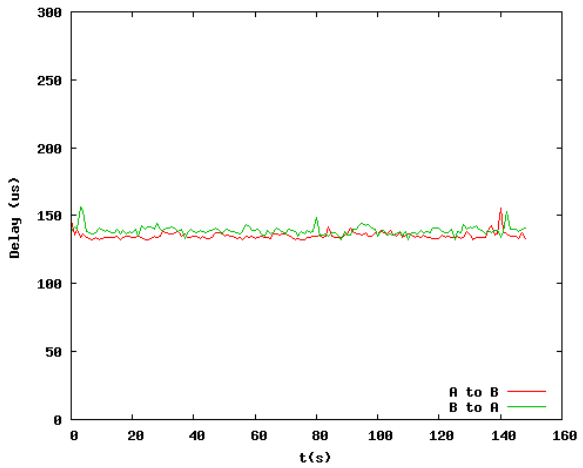

Abbildung 6: Messung mit Smartbits.

ziemlich konstant.

Abbildung 7 zeigt eine Messung zwischen zwei Hosts, die mit dem PPSKit synchronisiert werden. Die Messung beginnt genau in dem Moment, in welchem auf beiden Hosts der NTP-Daemon gestartet wird. Die Synchronisation ist also zu Beginn immernoch im Gange. Deutlich lässt sich erkennen, dass die Messung erst nach ca. 30 Minuten plausible Resultate liefert. Ausserdem ist eine negative Korrelation der beiden Kurven festzustellen. Diese lässt sich damit erklären, dass die Uhren in beiden Hosts jeweils mit der Zeit ein wenig gegeneinander verschoben (sie *driften*), dann aber von `ntpd` wieder sanft korrigiert werden.

Demgegenüber liefert die Messung mit der PPSClock (Abb. 8) bei gleichem physikalischen Aufbau bereits von Beginn weg gute Resultate. Es

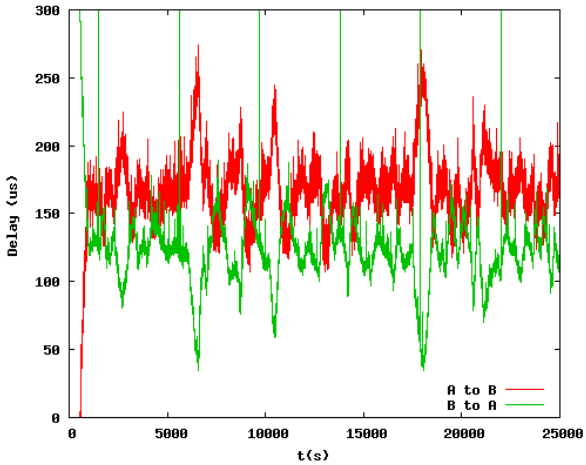

Abbildung 7: Messung der One-Way-Delays über zwei Hops in einem LAN. Die Synchronisation der Endgeräte erfolgt mit PPSKit.

ist aber auch ein etwas höherer Jitter ersichtlich. Die Abweichungen bewegen sich jedoch in einem Rahmen (ca. 40µs), welcher auf grösseren Distanzen mit üblicherweise markant grösseren One-Way-Delays nur von geringer Relevanz ist.

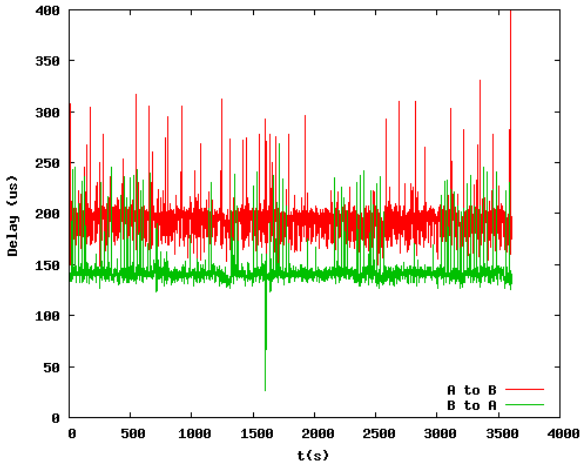

Abbildung 8: Synchronisation mit der PPSClock

Weiter sind in den Abbildungen 7 und 8 jeweils Unterschiede in den gemessenen Zeiten für Hin- und Rückweg zu erkennen – dies im Gegensatz zu der Refenenzmessung. Die Tatsache, dass diese Abweichungen aber bei beiden Uhren – PPSClock und PPSKit – erscheint, lässt die Vermutung zu, dass dies ein Problem der User-Applikation `gping` ist. Aus irgendwelchen Gründen scheint die Applikation die Messung des Weges von A nach B etwas zu verzögern.

4.4 Messung im Internet

Nachdem in Abschnitt 4.3 gezeigt wurde, dass eine Messung von One-Way-Delays mittels der vorgeschlagenen Ansätze grundsätzlich möglich ist und Werte liefert, die im Bereiche der Referenzmessung liegen, soll nun eine Messung auf einer etwas grösseren geographischen Distanz durchgeführt werden.

4.4.1 Versuchsaufbau

Die Messung findet zwischen einem Host in Brügg (BE) und einem in Bern statt. Die geographische Distanz zwischen den Geräten beträgt ca. 35 km, die Distanz im Netzwerk ist insgesamt 5 Hops. Beide Hosts sind durch einen ADSL-Router mit dem Internet verbunden. Auf beiden Hosts kommt das PPSKit zur Synchronisation zum Einsatz.

4.4.2 Resultate

Abbildung 9 zeigt die One-Way-Delay-Messung unter dem unter 4.4.1 erwähnten Versuchsaufbau. Die erste halbe Stunde ist, wie bei der PPSKit-Messung im LAN, gekennzeichnet durch eine Phase, in der keine brauchbaren Messungen möglich sind. Es sind aber auch danach – dies im Unterschied zur vorherigen Messung – deutlich Zeitspannen zu erkennen, in denen offensichtlich falsche (z.T. negative) One-Way-Delay-Werte gemessen wurden. So z.B. zwischen $t = 2'200$ und $t = 3'500$. Die Messungen bewegen sich aber im Grossen und Ganzen auf einer Linie um 15-20'000 Mikrosekunden.

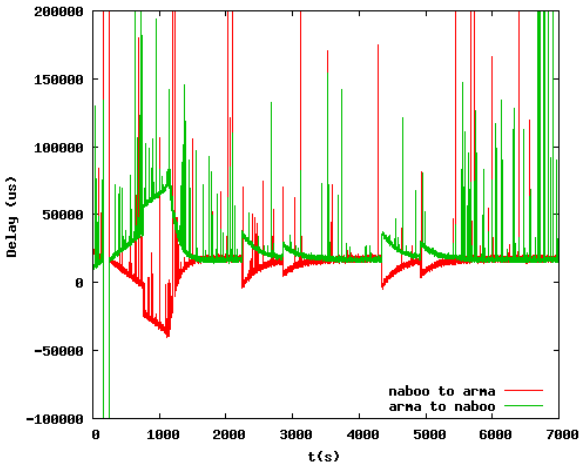

Abbildung 9: One-Way-Delay-Messung. Distanz zwischen den Hosts: ~35 km.

5 Zusammenfassung

Diese Arbeit liefert einen Vorschlag zur Bestimmung von One-Way-Delays zwischen weit entfernten Hosts mittels GPS-Empfangsgeräten. Eine One-Way-Delay-Messung erfordert exakt sychronisierte Uhren in den Endsystemen. Es werden zwei Ansätze zur Synchronisation vorgestellt: a) Eine neue Implementation einer genauen Uhr (PPSClock) mit einem relativ einfachen Zeitmodell, und b) NTP in Verbindung mit dem Kernelpatch PPSKit. Für One-Way-Delay-Messungen werden nur Standard-PCs mit serieller Schnittstelle sowie relativ preisgünstige GPS-Empfangsgeräte verwendet. Ein Einsatz ist somit praktisch überall ohne grossen Aufwand möglich und die Investitionen für Hardware sind gering. Diese Flexibilität erkauft man sich aber mit einer nur beschränkten Genauigkeit. Eine genaue Uhr stellt relativ hohe Anforderungen an die Echzeitfähigkeit eines Systems. Der Standard-PC mit Linux ist nicht *speziell* als Echtzeit-System ausgelegt und setzt dem ganzen Unterfangen deshalb Grenzen. Die durchgeführten Evaluationen erlauben aber den Schluss, dass sich beide vorgestellten Ansätze zur Zeitsynchronisation grundsätzlich für die One-Way-Delay-Messung eignen und Messwerte liefern können, die sich innerhalb weniger zehn Mikrosekunden bewegen.

6 Referenz

[1] www.colorado.edu/geography/gcraft/notes/gps/gps_f.html
[2] www.gpsclock.com
[3] RFC 2783, „Pulse per Second API for UNIX-like Operating Systems“, D. Mills et al, 2000
[4] A. Rubini et al, „Linux Device Drivers“ 3rd ed., O'Reilly Media 2005
[5] RFC 1589, „A Kernel Model for Precision Timekeeping“, D. Mills, 1994
[6] www.kernel.org
[7] „Linux Daemon Writing HOWTO“, May 2004
[8] Michael Tischer, „PC intern 5. Systemprogrammierung“, Data Becker 1995
[9] www.ntp.org
[10] www.nmea.org/pub/0183/
[11] Mills, D.L., "Improved algorithms for synchronizing computer network clocks", 1995

Anhang A - Der PPSKit-Kernel-Patch

Das PPSKit erfüllt zwei Aufgaben. Einerseits stellt es eine neue Kerneluhr mit Nanosekundenauflösung und zwei neuen Systemcalls nach RFC 1589 zur Verfügung, andererseits bietet es eine Implementation der in RFC 2783 vorgeschlagenen PPS API. Sowohl die neue Kerneluhr als auch PPS API sind tiefgreifende Änderungen im Standard Linux Kern. Das PPSKit ist deshalb nur als Kernel-Patch erhältlich. Der Kern muss nach dem Patchen neu übersetzt werden.

Voraussetzungen

Zuerst muss eine Umgebung geschaffen werden, in der sich der Linux Kern übersetzen lässt. Grunsätzlich sollte jede Standardinstallationen einer Linux-Distribution dafür geeignet sein. Es müssen einzig die Kernelsourcen runtergeladen und entpackt werden. Wir verwenden hier den Kernel 2.4.29 mit dem ensprechenden Patch PPSkit-2.1.5. Beide Pakete können von www.kernel.org bezogen werden.

Installation.

Zur Intsallation wechselt man in das Kernelsource-Verzeichnis und wendet den Patch an.

```
# cd /usr/src/linux-2.4.29-NANO
# patch -p1 < /PPSKit-2.1.5/patch-2.4.29
```

Bei der anschliessenden Kernelkonfiguration müssen die Optionen für NTP- und PPS-Support gemäss der Installationsanleitung gesetzt werden. Es folgen die üblichen Schritte für die Übersetzung des Kerns.

```
# make dep && make bzImage && make \
       modules && make install && make \
       modules_install
```

Damit andere Programme von der neuen PPS API profitieren können, müssen noch die zwei Header-Dateien `timex.h` und `timepps.h` an die richtige Stelle kopiert werden.

```
# cd /usr/src/linux-2.4.29-NANO/
# cp include/linux/timex.h /usr/include/sys/
# cp include/linux/timepps.h /usr/include/sys/
```

Die Schnittstelle, an der das PPS-Gerät angeschlossen ist, sollte im Sinne eines zukünftigen Gebrauchs durch `ntpd` noch verlinkt werden.

```
# ln -s /dev/ttyS0 /dev/pps0
# ln -s /dev/ttyS0 /dev/gps0
```

Nach dem Neustart des Systems mit dem neuen Kern kann die Funktionsweise der PPS API und der neuen Kerneluhr mit `ppsctl` kontrolliert werden. Mit

```
# /PPSKit-2.1.5/ppsctl -p/dev/pps0 -ea -Fad -m -ta
```

erhalten wir jede Sekunde einen Wert, der die Zeitdifferenz zum vorhergehenden PPS-Signal angibt. Er sollte nur sehr wenig von einer Sekunde abweichen. Die Installation des PPSKit ist nun abgeschlossen.

Anhang B – NTP-Installation und -Konfiguration

Voraussetzungen

- Download des NTP-Pakets (für GPS-Clock200 mindestens in der Version 4.2.0) von www.ntp.org.
- Installiertes PPSKit. Es werden vorallem die Dateien `/usr/include/sys/{timex.h, timepps.h}` benötigt.

Kompilation

Zuerst muss der Build-Prozess konfiguriert werden. Dies geschieht mit

```
# cd ntp-4.2.0
# ./configure
```

Soll NTP später in einem anderen als dem Standardverzeichnis installiert werden, so kann `configure` ein zusätzliches Argument übergeben werden. Z.B.:

```
# ./configure --prefix=/home/knoppix/master-KNOPPIX
```

Nach diesem Schritt sollte unbedingt kontrolliert werden, ob `configure` die Dateien `timpps.h` und `timex.h` richtig erkannt hat. Wenn ja, wird die Unterstützung für die RFCs 1589 und 2783 eingebunden.

```
# make
```

erledigt dann den Build-Prozess.

Installation

```
# make install
```

installiert das NTP-Paket unter dem bei `configure` angegebenen Root-Verzeichnis. Wenn nichts angegeben wurde, so ist dies einfach /.

Kofiguration

Der NTP-Daemon `ntpd` wird mit einem Kofigurationsfile `/etc/ntp.conf` konfiguriert. Folgende Einstellungen werden benötigt, um einen Stratum-1-Server mit Unterstützung für NMEA-Sentences und PPS-Signal (PPS-Clock 200) aufzusetzen:

```
#/etc/ntp.conf – Sample conf file for GPSClock 200

peer 127.127.20.0 prefer minpoll 4 maxpoll 4 # NMEA
fudge  127.127.20.0 stratum 0   # name it as a good clock
fudge  127.127.20.0 time1 -1.0  # GPSClock reports the next second


peer 127.127.22.0 minpoll 4 maxpoll 4 # PPS
fudge  127.127.22.0 stratum 0   # name it as a good clock
fudge  127.127.22.0 flag2 0     # Use PPS Assert
fudge  127.127.22.0 flag3 1     # Use kernel PPS

driftfile /tmp/ntp.drift
```

Die Angabe der Zeitquellen erfolgt durch IP-Adressen. Im Falle der GPS-Clock kommen eine Art „virtuelle“ IP-Adressen zum Einsatz: 127.127.20.0 ist z.B. die virtuelle Adresse für eine NMEA-Quelle am Gerät `/dev/gps0`; 127.127.22.0 bezeichnet eine PPS-Quelle an `/dev/pps0`.

Anhang C – Remastering von Knoppix 3.7 für den Gebrauch von NTP und GPSClock

Da der Aufbau eines Stratum-1-Servers mit Unterstützung für die GPSClock 200 (oder jeder anderen präzisen Zeitquelle) relativ aufwändig ist, wurde für diese Arbeit eine Knoppix-Live-CD entsprechend angepasst. Eine solche Live-CD ist ein Gewinn an Flexibilität im Umgang mit der GPS-Clock.
Für das Remastering wird Knoppix 3.7 verwendet. Diese Version basiert noch auf der alten 2.4-Linie des Kernels, alle neueren Versionen verwenden standardmässig den Kernel 2.6. Ein Austausch des Kernels lässt sich somit mit vertretbarem Aufwand bewerkstelligen.
Der eigentliche Prozess des Remasterings ist auf www.knoppix.net/wiki/Knoppix_Remastering_Howto_Deutsch ausführlich beschrieben. Hier seien nur die Schritte erwähnt, die speziell für eine NTP-Installation benötigt werden.

Voraussetzungen

- Knoppix 3.7 CD.
- Lauffähiges Linux-System mit cloop-Unterstützung.
- Ca. 2 GB freier Festplattenspeicher.
- Übersetzter Linuxkern 2.4.29 mit PPSkit-Patch (linux-2.4.29-NANO).
- Übersetztes cloop-Modul für den neuen Kern 2.4.29-NANO.

Anpassung des Root-Filesystems

Gemäss der Knoppix Remastering Howto nehmen wir an, die Dateien fürs Root-Filesystem befinden sich in `/mnt/hda4/knxsource/KNOPPIX`.

- Die Kompilation von NTP muss unter einem System geschehen, welches über ein installiertes PPSKit verfügt. Wichtig ist, dass die zwei unter Anhang A erwähnten Header-Dateien vorhanden sind.
- Weiter wird NTP gem. Angang A installiert. Als Root-Verzeichnis muss `/mnt/hda4/knxsource/KNOPPIX` angegebenen werden:
  ```
  # ./configure --prefix=/mnt/hda4/knxsource/KNOPPIX
  ```
- Zwei Gerätedateien müssen neu erstellt werden:
  ```
  # cd /mnt/hda4/knxsource/KNOPPIX/dev
  # ln -s ttyS0 pps0
  # ln -s ttyS0 gps0
  ```
- Die Konfigurationsdatei `ntp.conf` muss unter `/mnt/hda4/knxsource/KNOPPIX/etc/` neu erstellt und gem. Anhang A editiert werden.
- Weil ein neuer Kern zum Einsatz kommen soll, müssen auch noch die kompilierten Treiber bzw. Module an die richtige Stelle kopiert, sowie einige Links angepasst werden. Am besten bemüht man dazu die Installationsskripts des Kernels:
  ```
  # cd /usr/src/linux-2.4.29-NANO
  # make INSTALL_PATH=/mnt/hda4/knxsource/KNOPPIX install
  # make INSTALL_MOD_PATH=/mnt/hda4/knxsource/KNOPPIX modules_install
  ```

Anpassung der initrd

Der neue Kern macht eine Anpassung der Init-RAM-Disk unumgänglich. Die `initrd` befindet sich auf der Knoppix-CD unter `boot/isolinux/minirt24.gz`. Nach dem Entpacken und Mounten als Loop-Device, kann auf den Inhalt der Init-RAM-Disk zugegriffen werden. In der `initrd` müssen alle Module des alten Kerns durch diejenigen von 2.4.29-NANO ersetzt werden. Folgendes Script sollte dies bewerkstelligen. Natürlich muss es an die jeweilige Umgebung angepasst werden. Hier wird angenommen, dass die `initrd` auf `./master-minirt/` gemountet ist:

```
#!/bin/sh

EX_PATH=$(pwd)

# installation des cloop-moduls
cp -f ${EX_PATH}/sw/cloop-2.01.5/cloop.o master-minirt/modules/

SCSI_MODULES="aic7xxx.o aic7xxx_old.o BusLogic.o \
ncr53c8xx.o NCR53c406a.o initio.o mptscsih.o advansys.o aha1740.o aha1542.o
```

```
aha152x.o atp870u.o dtc.o eata.o fdomain.o gdth.o megaraid.o pas16.o pci2220i.o
pci2000.o psi240i.o qlogicfas.o qlogicfc.o qlogicisp.o seagate.o t128.o tmscsim.o
u14-34f.o ultrastor.o wd7000.o a100u2w.o 3w-xxxx.o"
cd ${EX_PATH}/sw/kernel/linux-2.4.29-NANO

# SCSI-Module, die in linuxrc erwähnt sind kopieren
for f in $SCSI_MODULES; do
       f_list=$(find . -name $f)
       for k in $f_list; do
              echo copying $k
              cp -fp $k ${EX_PATH}/master-minirt/modules/scsi/
       done
done

cd ${EX_PATH}
```

Anpassung des Bootvorgangs

Der alte Kernel muss noch durch den neuem ersetzt werden:

```
# cp /usr/src/linux-2.4.29-NANO/arch/i386/boot/bzImage \
       /mnt/hda4/knxmaster/boot/isolinux/linux24
```

Zu guter Letzt muss noch die Knoppix-CD zusammengebaut werden. Dieser Schritt ist in der Remastering-Howto beschrieben.