

Implementierung einer WLAN-Uhrensynchronisation mit RT-Linux

Georg Lukas
Laborpraktikum
AG Echtzeitsysteme und Kommunikation
Universität Magdeburg

2004-11-09

Zusammenfassung

In diesem Dokument wird die RT-Linux-spezifische Implementierung des in der Diplomarbeit von Spiro R. Trikaliotis[1] entwickelten und vorgestellten Uhren-Synchronisationsprotokolls beschrieben. Dieses auf drahtlose Netzwerke orientierte Protokoll stellt unter seinen Teilnehmern eine synchronisierte globale Uhr mit guter Präzision bereit.

Neben den internen Details der Implementierung für RT-Linux liegt das Augenmerk auch auf der Einbindung dieses Codes in eigene Anwendungen.

Inhaltsverzeichnis

1	Einleitung	3
1.1	Motivation	3
1.2	Aufgabenstellung und Ziele	3
2	Kontext der Arbeit	3
2.1	Uhren-Synchronisationsprotokoll	4
2.2	RT-Linux	4
2.3	RT-Orinoco	4
3	Implementierung	5
3.1	Strukturanpassung von RT-Orinoco	5
3.2	Zeitstempel-Übermittlung	7
3.3	Berechnung der globalen Client-Uhr	8
3.4	Ratenanpassung	9
3.5	Verwendung von Assembler	10
4	Schnittstelle für Anwendungen	11
5	Messung und Ergebnisse	12
5.1	Messaufbau	12
5.2	Messungen	13
5.3	Präzision ohne Indikation	14
5.4	Präzision mit Indikation	15
5.5	Besonderheiten	16
6	Zusammenfassung	18
A	Quelltext-Dateien	19
B	Eingesetzte Versionsnummern	20
C	Modulparameter	20
	Literatur	20

1 Einleitung

Im Rahmen seiner Diplomarbeit[1] hat Spiro R. Trikaliotis ein Uhrensynchronisationsprotokoll für drahtlose Netzwerke vorgestellt, und dieses als Gerätetreiber für Windows NT implementiert und getestet. Dieses Protokoll nutzt die Eigenschaften von 802.11[2]-basierter WLAN-Hardware, um darauf eine möglichst geringe *precision* der Teilnehmer-Uhren zu erreichen. Mit dieser Implementierung wurden maximale Abweichungen zwischen den Teilnehmern von unter $150\mu s$ erreicht.

1.1 Motivation

Da dieses Protokoll betriebssystemunabhängig entwickelt wurde, steht einer Portierung auf andere Plattformen nichts im Wege. Eine solche Portierung bietet sich insbesondere für RT-Linux an, das nicht nur die Voraussetzungen für harte Echtzeitanforderungen mitbringt, sondern auch einen geringeren Netzwerkstack-Overhead hat als Windows NT, wo zwischen Treiber und Hardware eine zusätzliche Abstraktionsschicht existiert.

Außerdem existieren unter RT-Linux zahlreiche Anwendungen, die von einer präzisen globalen Uhr profitieren würden, so zum Beispiel die Sensorfusion mehrerer Messstationen, wo für ein umfassendes Abbild der Umwelt nicht nur die Sensorwerte sondern auch der genaue Zeitpunkt, wann sie aufgenommen wurden, wichtig ist.

1.2 Aufgabenstellung und Ziele

Die diesem Dokument zugrunde liegende Arbeit erfolgte mit dem Ziel, das in [1] vorgestellte Synchronisationsprotokoll unter RT-Linux zu implementieren. Dabei sollte zum einen die bestmögliche Präzision anvisiert werden, und zum anderen die Modifikationen in RT-Orinoco gering bleiben, um die unabhängige Wartbarkeit zu erleichtern.

2 Kontext der Arbeit

Dieser Abschnitt soll einen kurzen Einblick in die drei Werke, auf denen die Implementierung aufbaut, gewähren. Für tiefere Auseinandersetzung sei hier auf das Kapitel 4.3 von [1] und die den Quellcode-Paketen beiliegende Dokumentation verwiesen.

2.1 Uhren-Synchronisationsprotokoll

Das Synchronisationsprotokoll arbeitet rundenbasiert. Es gibt einen Master und eine beliebige Anzahl Clients, die die Master-Uhr übernehmen. In jeder Runde gibt es ein Ereignis, das von allen Stationen möglichst zeitgleich erfasst wird – auf dem Medium WLAN wird dafür ein spezielles Broadcast-Paket verwendet. Dabei erfasst der Master den Versandzeitpunkt und die Clients den Empfang des Pakets.

Alle Stationen führen eine Liste von lokalen Zeitstempeln zu jedem Synchronisationspaket der letzten Runden, und in jeder Runde verschickt der Master seine jeweils aktualisierte Liste per Broadcast an alle Clients. Auf diese Weise können die Clients eine Korrelation zwischen den lokalen und den Master-Zeitstempeln herstellen und daraus die Master-Uhr herleiten.

Da auf diese Weise in jeder Runde eine abweichende Sicht der Clients auf die Master-Uhr entstehen kann, wird eine virtuelle Uhr eingeführt, die eine stetige Ratenanpassung an die zuletzt errechnete Master-Uhr durchführt. Diese wird dann der Anwendung zur Verfügung gestellt.

Die Firmware der verwendeten Karten macht es jedoch nicht möglich, den Versandzeitpunkt auf dem Master präzise zu erfassen. Daher kann eine weitere Station, der sog. Indikationsserver, hinzugezogen werden, um jede Runde mit einem Indikationspaket einzuleiten. Der Empfang dieses Pakets wird dann sowohl bei Master als auch bei den Clients als Grundlage für die Erfassung der Zeitstempel verwendet, was die Präzision deutlich verbessert.

2.2 RT-Linux

RT-Linux ist eine von FSMLabs[4] entwickelte Erweiterung für den Linux-Kernel, die Interrupt-Behandlung und Scheduling übernimmt und so harte Echtzeitsysteme mit einem „im Hintergrund“ laufenden Linux-Kernel und -Userland möglich macht. RT-Anwendungen werden in Form von Kernelmodulen geladen, die Kommunikation zwischen RT- und Userspace-Prozessen findet über FIFOs statt. In der hier vorgestellten Arbeit wurde die von FSM-Labs bereitgestellte quelloffene „RTLINUXFree“-Variante verwendet.

2.3 RT-Orinoco

Der für RT-Linux gewählte Ansatz erlaubt es nicht, gewöhnliche Linux-Treiber für Echtzeitaufgaben einzusetzen. Daher müssen die für eine Aufgabe benötigten Treiber auf RT-Linux portiert werden. RT-Orinoco[3] stellt eine solche, von Sebastian Vandersee durchgeführte, Portierung des *orinoco_cs*-Treibers dar.

3 Implementierung

3.1 Strukturanpassung von RT-Orinoco

Der RT-Orinoco-Treiber bildet die Schnittstelle zwischen RT-Modulen, Linux-Anwendungen und der Firmware von Intersil/Orinoco-Karten. Dabei kann er nicht wie ein herkömmliches Linux-Netzwerkdevice angesprochen werden, sondern stellt drei FIFOs (Paketempfang, RT-Steuerung und User-space-Steuerung) bereit. Über die Steuer-FIFOs können Ethernet-Pakete verschickt oder die Karte reinitialisiert werden.

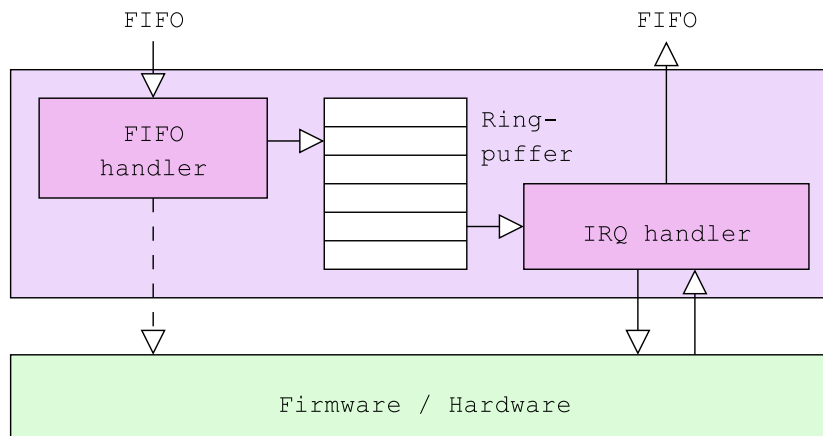


Abbildung 1: Originalstruktur des RT-Orinoco-Treibers

Der Treiber besitzt einen Send-Ringpuffer, in dem die Pakete vor dem Versand zwischengespeichert werden. Er dient dazu, Sendebursts abzufangen, wenn die Anwendung nacheinander mehrere Pakete versendet, und die Karte diese nicht mit der gleichen Geschwindigkeit abarbeiten kann.

Das Senden eines Pakets kann an zwei Stellen initiiert werden – ist die Karte gerade nicht beschäftigt und kommen Daten über eine der beiden Steuer-FIFOs rein, wird in `fifo_handler()` das Paket in den Ringpuffer geschrieben und sofort zum Versand an die Karte übergeben. Ist die Karte dagegen gerade aktiv, so landet das Paket nur zur späteren Abarbeitung im Puffer. Wenn die Karte durch einen IRQ bekannt gibt, dass sie wieder Pakete annehmen kann, überprüft `rt_orinoco_interrupt()`, ob im Sendepuffer weitere Pakete vorliegen, und schickt diese gegebenenfalls an die Firmware.

Für eine Uhrensynchronisation ist der Ringpuffer jedoch nachteilhaft – ein voller Sendepuffer führt zu einer erhöhten Latenz beim Versand der Pakete, was zwar keine direkte Auswirkungen auf die Präzision des Protokolls hat, allerdings bei einer Verzögerung über das Rundenende hinaus als Paketverlust

bemerkbar wird.

Um dieses Problem zu umgehen wurde ein Out-Of-Band-Sendepuffer für ein einzelnes Paket (`oobtxbuf`) eingeführt, der Vorrang vor dem regulären Ringpuffer hat. Befindet sich darin ein Paket, wird dieses verschickt sobald die Karte bereit ist. Ist der OOB-Puffer leer, so werden Daten aus dem normalen Ringpuffer versendet. Für diese Überprüfung wurde die Funktion `rt_orinoco_try_to_xmit_next()` eingeführt, die von FIFO- und Interrupt-Handler aufgerufen wird.

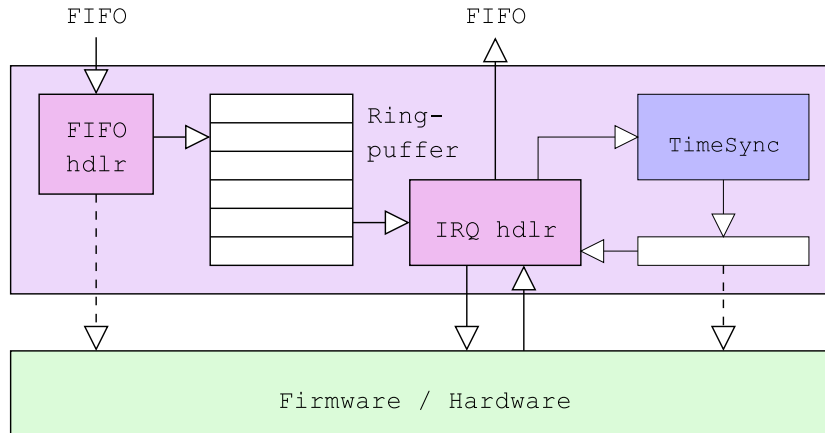


Abbildung 2: Erweiterung von RT-Orinoco zur Uhrensynchronisation

Der Interrupt-Handler ist für eine weitere synchronisationsbezogene Aufgabe zuständig. Da die Auswertung des IRQs einige Zeit in Anspruch nimmt, und im Fall des Paketempfangs das Paket erst noch von der Karte kopiert werden muss, bevor festgestellt werden kann ob es sich um ein Synchronisationsframe handelt, wird an erster Stelle in `rt_orinoco_interrupt()` ein Zeitstempel genommen, der an die Event-Handler, und von da an den Synchronisationscode weitergegeben wird. Das erlaubt trotz eines langen Code-Pfads den IRQ-Zeitpunkt und das entsprechende Paket zusammen zu betrachten.

Die Empfangsroutine (`__rt_orinoco_ev_rx()`) ruft grundsätzlich für jedes Paket `ts_analyzeframe()`, das dann anhand des Paketinhalts entscheidet, ob es sich um ein Synchronisationsdatagramm handelt oder nicht und es gegebenenfalls auswertet. Daraufhin teilt die Funktion dem Aufrufer mit, ob das Paket bearbeitet wurde und verworfen werden kann, oder ob es an die Empfangs-FIFO weitergegeben werden soll. Pakete mit dem für die Uhrensynchronisation definierten Ethernet-Frame-Typ `TS_ETH_PACKETTYPE (0xF00D)` an die Broadcast-Zieladresse werden grundsätzlich als „bearbeitet“ quittiert, auch wenn die Uhrensynchronisation auf dem empfangenden System nicht aktiviert ist. Dadurch soll vermieden werden, dass die Anwendung

durch fremde Synchronisationspakete verwirrt wird.

Weitere Eingriffe in den Code sind die Aufrufe von `ts_configure()` beim Initialisieren von RT-Orinoco, `ts_start()` beim Einbuchten in ein drahtloses Netzwerk und `ts_stop()` beim Disassoziieren bzw. beim Entladen des Moduls. Mit den letzten beiden Funktionen wird auch der Ereignisthread bei Indikator/Master gestartet und beendet, der für das Generieren der Indikationspakete zuständig ist.

3.2 Zeitstempel-Übermittlung

Auf dem Master dient die lokale Uhr gleichzeitig als globale, die Clients berechnen dagegen die globale Uhrzeit aus den Synchronisationspaketen des Masters und den lokalen Empfangszeitstempeln.

Für die Ermittlung der lokalen Zeit dient das in `ts.h` definierte Makro `ts_localtime()`, welches im Standardfall auf die RT-Linux-eigene Funktion `gethrtime()` abgebildet ist. Sollte ein anderer Zeitgeber benötigt werden, empfiehlt es sich, das Makro entsprechend anzupassen.

Die Erfassung aller Zeitstempel erfolgt im Interrupt-Handler des RT-Orinoco-Treibers, `rt_orinoco_interrupt()`, wo auch das `ts_localtime()`-Makro verwendet wird. Die so gewonnenen Zeitstempel tragen sowohl Master als auch Clients über mehrere Runden in die Tabelle `ind_timestamps[]` ein, und der Master verschickt die Rundenummer und die Zeitstempel der Vorrunden in einem Ethernet-Frame des Typs `TS_ETH_PACKETTYPE` an alle Clients.

Geht ein Paket verloren, erkennt eine Station das erst beim Empfang des Datagramms der nächsten Runde, da nur der Master (respektive nur der Indikationsserver) einen eigenen periodischen Thread hat, und alle Clients event-basiert agieren. Für nicht erfasste Zeitpunkte tragen diese dann `T_TIME_INVALID` in die Liste ein.

Die Clients haben neben dieser Liste der lokalen Zeitstempel auch `timepairqueue[]`, eine Queue, in der den lokalen Zeitstempeln die des Masters zugeordnet sind. Diese dient beim Bilden der globalen Uhr dazu, zwei zeitlich weit entfernte Messpunkte zu verwenden, um den Einfluss von Störungen zu minimieren.

Jedes Mal beim Empfang eines Master-Pakets werden in `ts_client_get_master()` die Paare gebildet und mit `ts_add_timepair()` in `timepairqueue[]` abgelegt. Sobald die Queue zwei Paare enthält, berechnet der Client die virtuelle Uhr, und meldet dass er synchronisiert ist. Mit jedem neuen Queue-Eintrag steigt der Abstand zwischen den verwendeten Werten und der Einfluß von Messfehlern sinkt.

3.3 Berechnung der globalen Client-Uhr

Aus Sicht des Clients ist die globale Uhr eine lineare Funktion über der lokalen (physischen) Uhr.

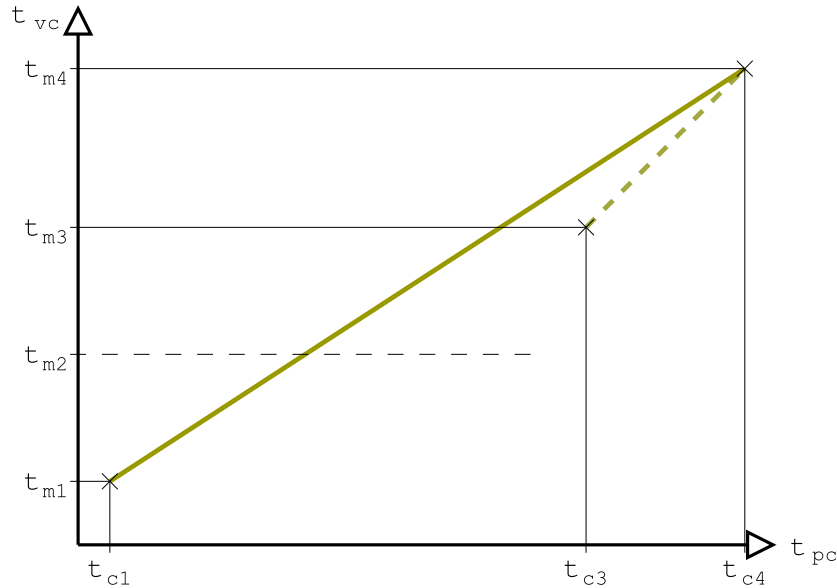


Abbildung 3: Sicht eines Clients auf die globale Uhr

Die Parameter einer solchen Uhr sind in `struct vclock (ts.h)` definiert. Dabei ist `gradient` der Anstieg, und `offset` ist der Abstand vom Nullpunkt bei $t_{pc} = 0$. Die Uhr wird aus dem ersten und letzten Wertepaar von `timepairqueue` berechnet:

$$\text{gradient} = \frac{t_{mN} - t_{m1}}{t_{cN} - t_{c1}} \quad (1)$$

$$\text{offset} = t_{m1} - t_{c1} \cdot \text{gradient} \quad (2)$$

Dabei steht t_{mX} in `timepairqueue[X].mastertime` und t_{cX} in `timepairqueue[X].clienttime`.

Es ist möglich, für die Berechnung mehr als nur den ersten und den letzten Wert der Queue heranzuziehen, allerdings ist es mit einem erhöhten Rechenaufwand verbunden, und bringt vermutlich keine deutliche Verbesserung gegenüber dem naiven Ansatz.

Die Berechnung der virtuellen Uhr aus der physischen erfolgt demnach wie folgt:

$$t_{vc} = \text{gradient} \cdot t_{pc} + \text{offset} \quad (3)$$

Da der Wert für den Anstieg in der Größenordnung von 1 ist, wäre zur Speicherung eine `float`-Variable zweckmäßig. Da Floating-Point-Berechnungen aber im Allgemeinen länger dauern und einen zusätzlichen Overhead für den Kernel bedeuten, wurde in dieser Implementierung stattdessen auf eine Fixpunkt-Notation zurückgegriffen. Die Variable `gradient` enthält den Anstieg entsprechend als 32.32-Fixpunkt-Wert.

3.4 Ratenanpassung

In jeder Runde wird die globale Uhr neu berechnet, um die Abweichung vom Master gering zu halten. Da dies unter Umständen zu einer Unstetigkeit führt, für die meisten Anwendungen aber eine Uhr ohne Zeitsprünge benötigt wird, findet eine Ratenanpassung statt.

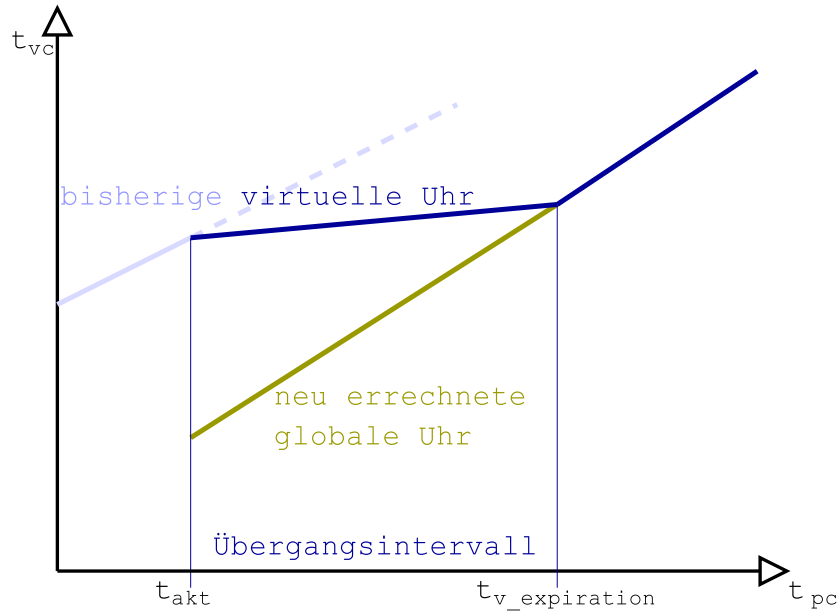


Abbildung 4: Verlauf der Uhr bei Ratenanpassung

Dazu wird eine Zwischenuhr eingeführt, die zum Zeitpunkt der Berechnung den selben Wert hat wie die bisherige virtuelle Uhr, und die zur errechneten globalen Uhr konvergiert. Wenn sie den Wert der globalen Uhr erreicht, hat sie ihre Funktion erfüllt, und von da an wird die globale Uhr verwendet.

Diese beiden Uhren und der (lokale) Zeitpunkt für die Umschaltung sind in `struct vclockpair (ts.h)` definiert. Die Zwischenuhr ist `struct vclock virt`, und die Master-Uhr `master` vom selben Datentyp. Vor dem Zeitpunkt in `v_expiration` ist die erste, danach die zweite Uhr gültig.

3.5 Verwendung von Assembler

Da die Zeitstempel in 64-Bittigen Variablen gespeichert werden, würde bei der Multiplikation des Zeitstempels mit `gradient` ein 96.32-Fixpunkt-Wert entstehen, von dem aber nur die unteren 64 Bits des ganzzahligen Anteils relevant sind. Da C keine Funktion zum Multiplizieren von zwei 64-Bit-Werten mit 128-Bit-Ergebnis bietet, wurde diese in Assembler nachentwickelt.

Die resultierende Routine `mul64x32_32()` nimmt zwei 64-Bit-Werte ohne Vorzeichen als Parameter und liefert einen 64-Bit-Wert zurück – nämlich die mittleren 64 Bit des 128-bittigen Resultats. Die unteren 32 Bits im Ergebnis – der Nachkommaanteil des virtuellen Zeitstempels – werden verworfen. Sind allerdings die oberen 32 Bits nicht 0, so wird das als Überlauf gewertet und die Funktion liefert den Maximalwert ($2^{64} - 1$) zurück.

Diese Funktion, deren Deklaration sich in `math64.asm` befindet, erfordert zum Kompilieren einen externen Assembler wie NASM[5].

Daneben befindet sich in `longlong.h` aus der GNU libc übernommener Assembler-Code zur 64-Bit-Division. Dieser Code wird beim Linken von Userland-Programmen automatisch eingebunden, muss bei Kernel-Modulen allerdings händisch dazugelinkt werden.

4 Schnittstelle für Anwendungen

Die API ist auf RT-Linux-Module ausgelegt. Zu ihrer Verwendung muss die Headerdatei `ts.h` eingebunden werden. Danach stehen dem Nutzer die Funktionen `ts_localtime()` zum Einholen der lokalen Zeit, `ts_issync()` zum Überprüfen, ob die virtuelle Uhr bereits synchronisiert ist, sowie `ts_virtual()` und `ts_physical()` zum Umrechnen der physischen Uhr in die virtuelle und zurück.

```
1 #include "ts.h"
2
3 int init_module(void) {
4     /* Prüfung der virtuellen Uhr */
5     if (!ts_issync()) {
6         rtl_printf("Initialisierungsfehler: Uhr noch
7                 nicht synchronisiert!\n");
8         return 1;
9     }
10    /* Eigene Initialisierung */
11    ...
12    /* Ereignis-Überwachung beginnen */
13    return rtl_request_irq(MY_IRQ, event_handler);
14 }
15
16 unsigned int event_handler(unsigned int irq,
17                            struct pt_regs *regs) {
18     T_TIME irqtime = ts_local();
19     T_TIME vrttime = ts_virtual(irqtime);
20     /* Normale IRQ-Behandlung */
21     ...
22     /* Mitteilung der Event-Zeit an andere Stationen */
23     broadcast_message(ALL, GOT_EVENT, vrttime);
24     ...
25     /* Aufräumen */
26     rtl_hard_enable_irq(MY_IRQ);
27     return 0;
28 }
```

Listing 1: Beispiel für die Uhrensynchronisation

An einem Beispiel soll die Funktionsweise vorgestellt werden. Beim Laden des Moduls wird zunächst (in Zeile 5) überprüft, ob die virtuelle Uhr synchronisiert und funktionsbereit ist. Ist das noch nicht der Fall, wird der

Ladevorgang mit einer Fehlermeldung abgebrochen (Zeilen 6 und 7). Im positiven Fall dagegen wird eine Funktion als IRQ-Handler eingetragen, die später bei externen Ereignissen vom RT-Linux-Kernel aufgerufen wird.

Tritt ein entsprechendes externes Ereignis auf, beispielsweise eine Pegeländerung am Parallelport, ruft der Kernel den dafür zuständigen IRQ-Handler (`event_handler()`, Zeilen 15-27 im Listing) auf. Dieser liest als Erstes mit `irqtime = ts_local()` die physische Uhr aus, und rechnet das Ergebnis dann mit `vrtime = ts_virtual(irqtime)` nach der gerade aktuellen virtuellen Uhr in die virtuelle Zeit um. Wenn die physikalische Uhrzeit in `irqtime` nicht benötigt wird, kann auch sofort `vrtime = ts_virtual(ts_local())` geschrieben werden.

Danach findet die für IRQ-Handler übliche Arbeit statt, so z.B. die Überprüfung, ob der IRQ überhaupt für dieses Modul bestimmt war, oder von einem anderen Gerät ausgelöst wurde. Außerdem kann hier das Ereignis ausgewertet und entsprechend darauf reagiert werden. Der Zeitpunkt des Ereignisses wird dabei über `vrtime` referenziert (Zeile 22).

Mit den letzten beiden Befehlen schließlich wird signalisiert, dass weitere IRQs akzeptiert werden können (Zeile 25), und dass dieser Interrupt erfolgreich abgearbeitet wurde (Zeile 26). Ab hier kann der Handler wieder vom Kernel aufgerufen werden.

5 Messung und Ergebnisse

5.1 Messaufbau

Der Messaufbau wurde dem in Kapitel 6 von [1] beschriebenen nachempfunden: vier Stationen nehmen über WLAN an dem Synchronisationsprotokoll teil, und sind dabei über ihre Parallelports mit dem Messrechner verbunden.

Der Messrechner entspricht dem in [1] verwendeten, so dass die dort durchgeführten Messungen der zeitlichen Auflösung übertragbar sein sollten, und eine Messgenauigkeit von ca $5\mu s$ zu erwarten ist.

Die an der Uhrensynchronisation beteiligten Stationen sind mit Athlon XP 2400+ CPUs und je 512MB RAM ausgestattet. Sie enthalten ferner Orinoco-Karten in PCMCIA-Adaptern.

Jede der vier Stationen hat Schreibzugriff auf drei Bits ihres Parallelports, die vom Messrechner in einer tight loop ausgelesen und auf Änderungen überwacht werden. Dabei wird bei jedem Hardware-IRQ von der Orinoco-Karte Bit 0 gekippt – zu diesem Zeitpunkt wird auch der lokale Zeitstempel genommen. Bit 1 wird in der Empfangsroutine getoggelt, damit man zwischen empfangenen Paketen und anderen Ereignissen unterscheiden kann.

Bit 2 schließlich wird verwendet, um den Grad der Synchronisation zu bestimmen. Dazu wird das Kernel-Modul `ts_test` verwendet, was von der virtuellen (bzw. auf dem Master von der physischen) Uhr ausgehend alle $2^{30}ns$ (also ca. im Sekundentakt)¹ sein Bit umkippt.

Als Kompromiss zwischen Genauigkeit und CPU-Auslastung wird eine Kombination aus Scheduling und Polling für die Annäherung an den Zeitpunkt verwendet - der Kernel weckt den zuständigen Thread alle $10ms$ auf, damit dieser nachschauen kann, ob der nächste Synchronisationszeitpunkt innerhalb der nächsten $10ms$ liegt. Ist das der Fall, wird mit einer busy loop darauf gewartet, dass der Zeitpunkt eintritt, und dann sofort das Bit getoggelt.

Bei Messungen ohne Indikation ist eine Station der Master, der die drei anderen als Clients synchronisiert, kommt der Indikationsserver dazu, können nur noch zwei Clients synchronisiert werden.

Aus der Gegenüberstellung der Bit-2-Werte der vier Stationen lässt sich schließlich die Präzision der globalen Uhr ersehen. Die im Folgenden gezeigten Diagramme enthalten auf der Y-Achse die Differenz der Client-Triggerzeitpunkte gegenüber dem Master, wobei alle Werte vom weiter oben beschriebenen Messsystem erfasst wurden. Es ist zu beachten, dass die Abweichung um mehrere Größenordnungen geringer ist als die auf der X-Achse abgetragene Zeit. Die nichtproportionale Darstellung wurde gewählt, damit die Diagramme aussagekräftig bleiben.

5.2 Messungen

Um neben der Präzision der Synchronisation unter Normalbedingungen auch die Auswirkungen eines Indikationsservers zu ermitteln, wurden insgesamt vier Arten von Messreihen durchgeführt: Jeweils mit und ohne Indikation sowie auf einem freien Medium und unter Last. Jede Messung lief mindestens fünf Minuten.

Die Belastung des Mediums wurde dadurch realisiert, dass zwei weitere in die selbe Ad-Hoc-Gruppe eingebuchte Rechner untereinander per `iperf -fA -d[6]` TCP-Traffic bideriktional mit insgesamt ca. $650KByte/s$ ausgetauscht haben.

Des Weiteren wurde exemplarisch geprüft, wie die Clients auf einen Ausfall der Synchronisation reagieren. Dazu wurde per `cardctl eject` auf dem Master das Versenden der Rundennachrichten unterbunden, die `ts_test`-Module liefen aber auf allen vier Rechnern weiter.

¹Der Wert 2^{30} erlaubt es, mit binären Operationen die Verwendung von kostspieligeren Modulo-Berechnungen zu ersetzen.

5.3 Präzision ohne Indikation

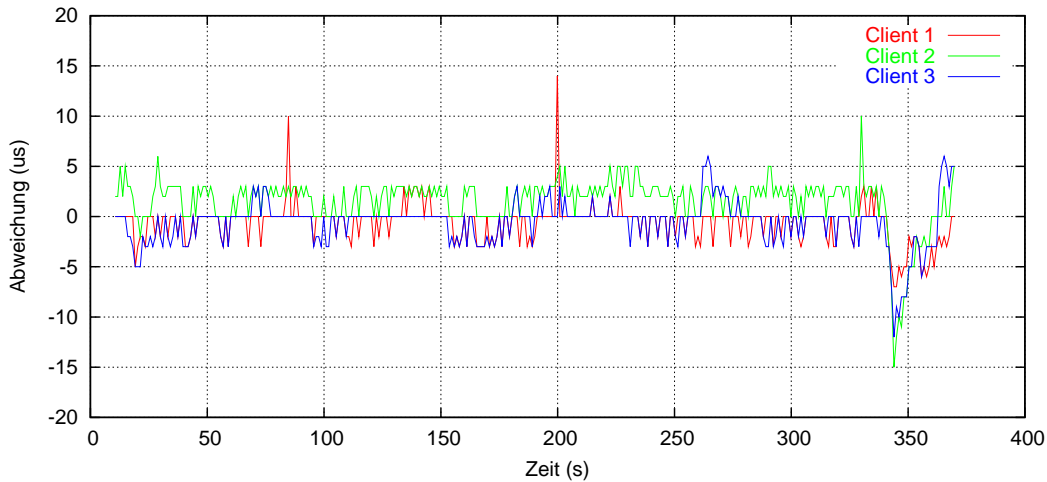


Abbildung 5: Synchronisation ohne Indikation ohne Last (mccc01)

Auf einem unbelasteten Medium pendelt sich die *precision* der Globalen Uhr ca. 30 Sekunden nach Beginn der Synchronisation in einem Korridor von $30\mu s$ ein (Abb. 5 und 6). Die meisten Werte liegen sogar näher als $5\mu s$ zum Master, befinden sich also im Rahmen der Messgenauigkeit.

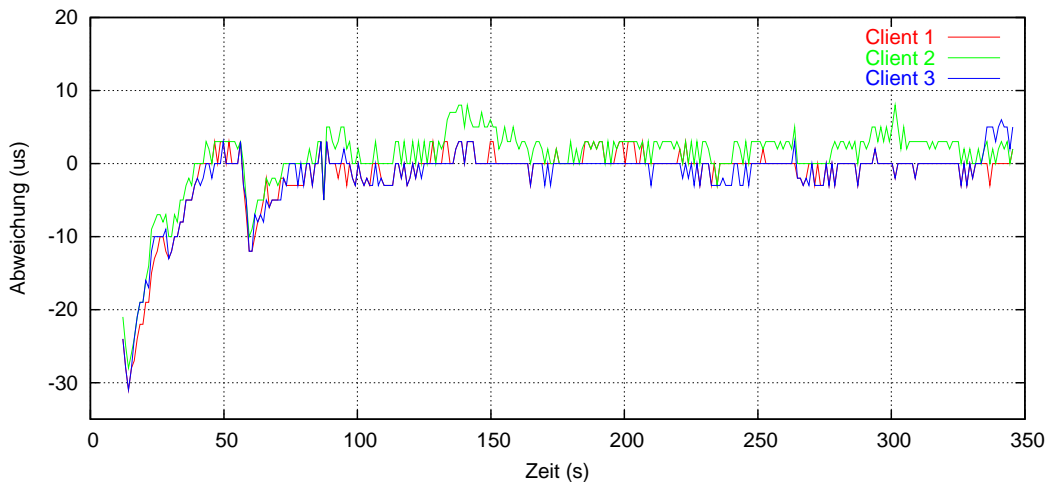


Abbildung 6: Synchronisation ohne Indikation ohne Last (mccc012)

Auffällig sind die Stellen, bei denen alle Clients um bis zu $15\mu s$ vorlaufen. Diese begründen sich dadurch, dass der Master seinen Versand-IRQ etwas später von der Hardware bekommt, als die Clients das Paket empfangen.

Dadurch versendet der Master bei der nächsten Runde irreführende Informationen, und alle Clients ziehen daraus den selben Schluss.

Wird das Medium belastet, tritt ein ähnliches Verhalten auf (Abb. 7). Die Häufigkeit der Sprünge aller Clients nimmt zu.

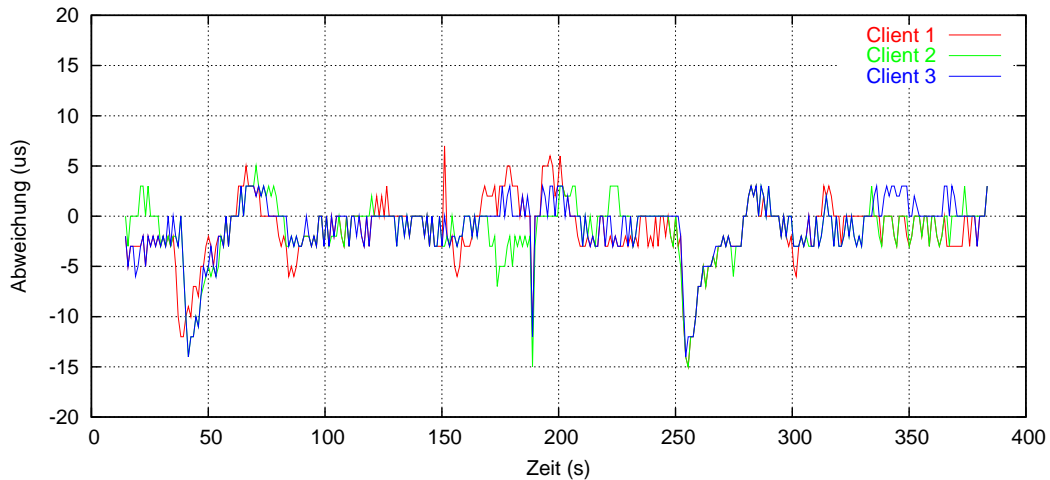


Abbildung 7: Synchronisation ohne Indikation unter Last (mccc650m)

5.4 Präzision mit Indikation

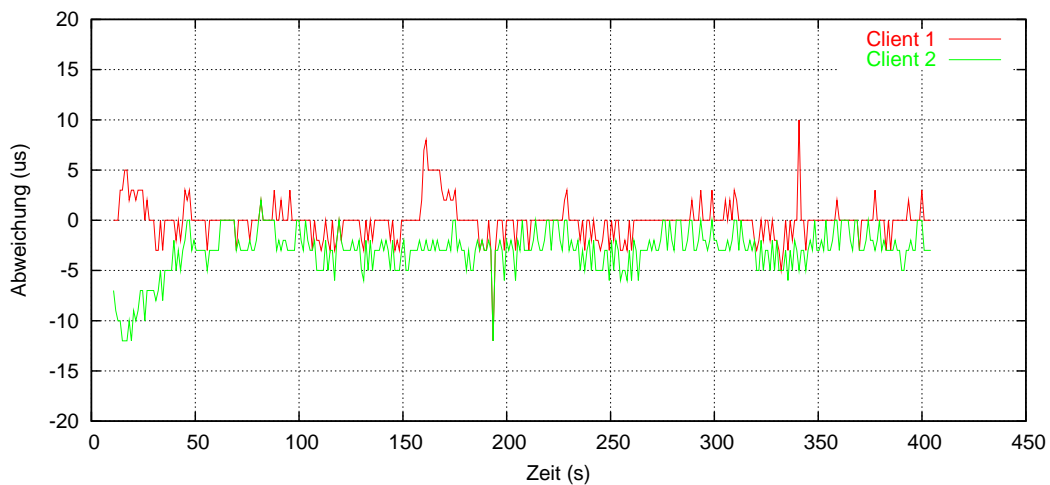


Abbildung 8: Synchronisation mit Indikation ohne Last (micc0l)

Bei der Verwendung eines Indikationsservers sind die Werte wieder mit wenigen Ausnahmen im Rahmen der Messgenauigkeit. Die bei den ersten

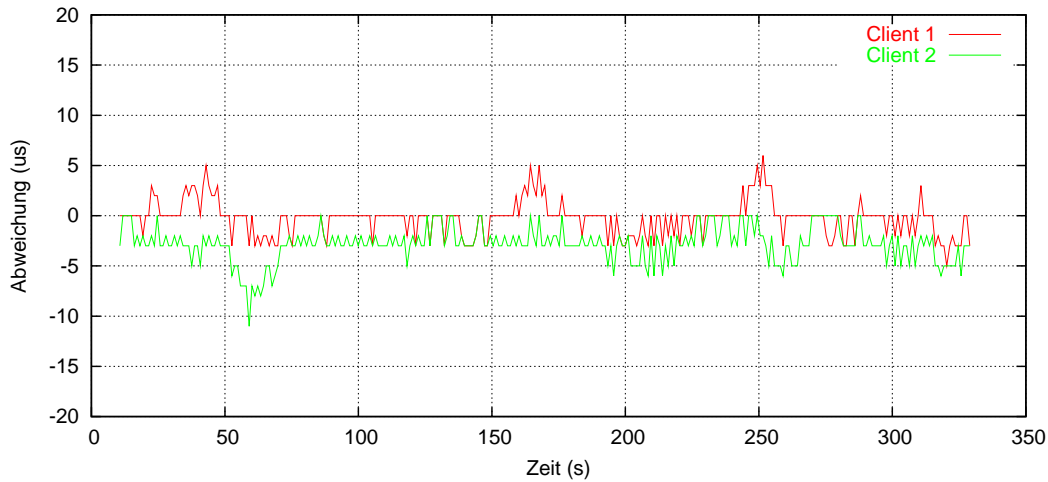


Abbildung 9: Synchronisation mit Indikation unter Last (micc650k)

Messreihen beobachteten Sprünge aller Clients treten allerdings nicht mehr auf - hier messen Master und die Clients wirklich das selbe Ereignis, was sich positiv auf die Ergebnisse auswirkt.

Bei einzelnen Clients treten jedoch auch manchmal Abweichungen vom Master um bis zu $12\mu s$ auf, so dass die effektive *precision* nicht wesentlich besser ist als ohne Indikation.

5.5 Besonderheiten

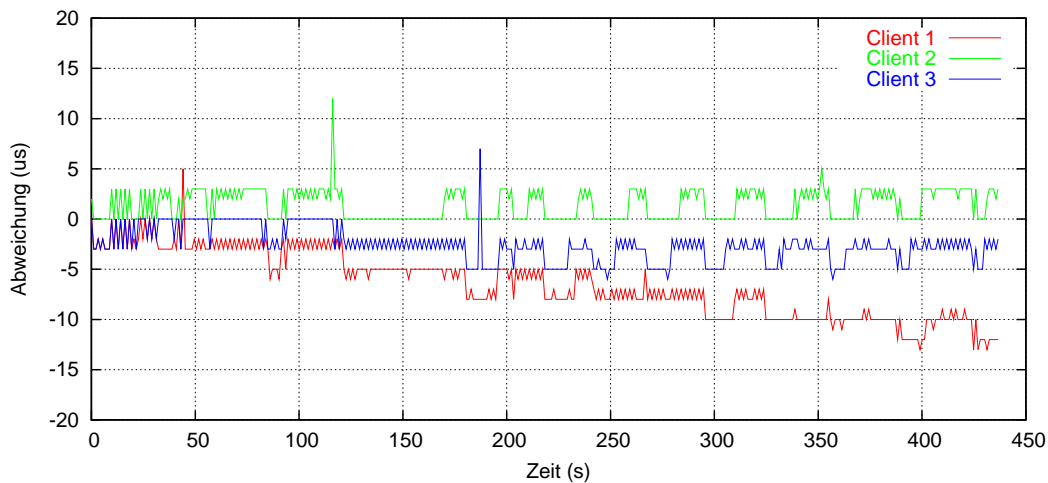


Abbildung 10: Synchronisationsausfall ohne Last (mcccxxx)

Fällt die Synchronisation aus irgendwelchen Gründen aus, driften die Clients langsam vom Master und voneinander weg, erreichen allerdings erst nach mehreren Minuten einen Abstand, der außerhalb der normalen *accuracy* liegt (Abb. 10 und 11). Da die Clients ihre virtuelle Uhr nach dem zuletzt vom Master empfangenen Paket ausrichten, kann es bei einem Paketverlust auch zu stärkeren Abweichungen einzelner Clients kommen.

Der Ausreißer am Ende von Abb. 11 scheint ein Problem des Messaufbaus zu sein, das beim Beenden der Messung auftrat - alle Clients haben genau $0\mu s$ Abstand zum Master, was darauf hindeutet, dass das Messprogramm durch einen IRQ ausgesetzt wurde, und erst nach dessen Abarbeitung alle vier Pegeländerungen als „gleichzeitig“ wahrgenommen hat.

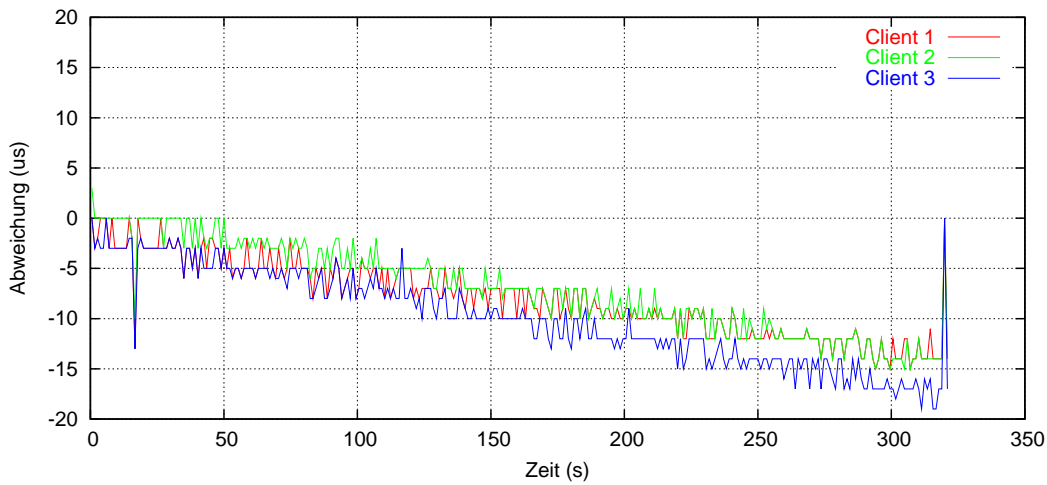


Abbildung 11: Synchronisationsausfall unter Last (mccc650x)

Weiterhin kam es bei allen Messungen sporadisch zu kurzen Positiv-Peaks auf Bit 2 von unterschiedlichen Clients. Da diese nur in eine Richtung und immer gleichzeitig zu Pegeländerungen auf den anderen Bits auftraten, ist von einem Übersprechen auf dem Parallelport auszugehen. Um diese Anomalie, die keine sichtbare Auswirkung auf die Präzision der Messungen hatte, auszugrenzen, wurde ein kurzes C-Programm (`parse.c`) geschrieben, welches alle Messwert-N-Tupel, die unter $30ms$ auseinander liegen (also einen potentiell deutlich höheren Abstand haben, als von den Messungen gezeigt), zu einer Gruppe zusammenfasst, und alle Werte, die keine vollständige Gruppe bilden, nur als Kommentar ausgibt.

6 Zusammenfassung

Das Synchronisationsprotokoll ist in seiner jetzigen Implementierung einsetzbar und liefert kurz nach Anfang der Synchronisation eine *precision* von $30\mu s$. Unter Einsatz eines Indikationsservers lässt sich dieser Wert geringfügig verbessern, was aber auf Kosten eines zusätzlichen Rechners im System und damit mit einer weiteren Fehlerquelle einhergeht.

Es ist denkbar, in einer hierauf aufbauenden Implementierung Master und Indikationsserver redundant auszustatten, oder bei Ausfall diese Aufgabe dynamisch einem der Clients zuzuweisen. Alternativ wäre es auch möglich, dass die Clients der Reihe nach Indikationspakete versenden, und so jeweils nur ein Event auslassen.

Außerdem können durch striktere Überprüfung fehlerhafte Messwerte ausgefiltert werden, um so die Präzision der globalen Uhr weiter zu verbessern.

Ferner sollte der Ansatz, die virtuelle Uhr für das Scheduling von lokalen Ereignissen zu verwenden, weiter verfolgt werden. Dabei ist allerdings zu beachten, dass durch die stetige Resynchronisation und durch die Ratenanpassung zu unterschiedlichen Zeitpunkten unterschiedliche virtuelle Uhren Gültigkeit haben.

A Quelltext-Dateien

<code>config/*</code>	pcmcia_cs-Konfigurationsdateien
<code>config.mk</code>	Konfiguration der Source-Pfade für Linux, RT-Orinoco und <code>pcmcia_cs</code>
<code>divdi3.c</code>	libc-C-Quellcode für 64-Bit-Division
<code>longlong.h</code>	libc-Assembler-Quellcode für 64-Bit-Division
<code>Makefile</code>	Übersetzungsanweisungen für <code>make</code>
<code>math64.asm</code>	Assembler-Quellcode für <code>mul64x32_32()</code>
<code>math64.h</code>	Header für <code>math64.asm</code>
<code>measure.c</code>	Symbolexport zur Präzisionsmessung
<code>measure.h</code>	Definitionen für die Präzisionsmessung über den Parallelport
<code>README</code>	Originalhilfe zum RT-Orinoco-Treiber
<code>README.ts</code>	Modulparameter für unterschiedliche Szenarios der Uhrensynchronisation
<code>rt_orinoco-config.h</code>	Betriebs- und Konfigurationsparameter für RT-Orinoco (insb. Typ Name, und Kanal des WLANs)
<code>rt_orinoco.c</code>	Kern des RT-Orinoco-Treibers
<code>rt_orinoco.h</code>	RT-Orinoco-eigene Deklarationen und Prototypen
<code>ts-internal.h</code>	Interne Datenstrukturen für die Uhrensynchronisation
<code>ts-rto.h</code>	Für die Nutzung in RT-Orinoco exportierte Funktionsprototypen
<code>ts.c</code>	Master, Client und Indikationsserver; Lesen der Konfiguration
<code>ts.h</code>	Header für die Verwendung in eigenen Anwendungen
<code>ts_test.c</code>	Test-Code zum Messen der Präzision über den Parallelport
<code>vclock.c</code>	Initialisierung der globalen Uhr; Umrechnung zwischen lokaler und globaler Uhr

B Eingesetzte Versionsnummern

Distribution	Debian 3.1 (Sarge)
Linux-Kernel	2.4.18-rtl31
RT-Linux	3.1
RT-Orinoco	0.7.4
pcmcia_cs	3.2.5
GCC	3.3.4

C Modulparameter

ts_type	i m c i m i c	„I“ndikator, „M“aster oder „C“lient (letzte optional mit Indikation)
ts_master	MAC-Adresse	Adresse des Masters (nur für Client nötig)
ts_indicator	MAC-Adresse	Adresse des Indikationsservers (für Master und Client mit Indikation nötig)
ts_interval	integer	Zeit zwischen Runden in <i>ms</i> (nur für Master bzw. Indikationsserver)

Literatur

- [1] „Uhrensynchronisation in einem lokalen Funknetzwerk“, Diplomarbeit von Spiro R. Trikaliotis; GMD Research Series, 2000, No. 5, Bonn
- [2] IEEE Working Group for WLAN Standards, <http://grouper.ieee.org/groups/802/11/>
- [3] RT-Orinoco, bisher unveröffentlicht
- [4] FSMLabs, Inc., <http://www.fsmlabs.com/>
- [5] The Netwide Assembler, <http://nasm.sourceforge.net/>
- [6] Iperf, <http://dast.nlanr.net/Projects/Iperf/>