Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Felix Michel, Philippe Wüger

Angewandte Uhrensynchronisation auf BTnodes

Studienarbeit SA-2005-02
Wintersemester 2004/2005

Betreuer: Lennart Meier

Verantwortlicher:
Prof. Dr. Lothar Thiele

4.2.2005

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK
Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Prof. Dr. L. Thiele

Wintersemester 2004/2005

Angewandte Uhrensynchronisation auf BTnodes

Studienarbeit

für
Felix Michel und Philippe Wüger

Betreuer: Lennart Meier
Ausgabe: 18. Oktober 2004
Abgabe: 7. Februar 2005

Einleitung

BTnodes sind kleine (6 x 4 x 0.5 cm^3), programmierbare Netzknoten; sie bestehen im Wesentlichen aus einem Mikrocontroller und einem Radiomodul, welches drahtlose Kommunikation gemäss dem Bluetooth-Standard [8] ermöglicht [3, 2].

In einem Netz von BTnodes ist die Synchronisation der Uhren der BTnodes wünschenswert. Sie ermöglicht einerseits das Zusammenfassen von verteilten Messergebnissen, andererseits kann die Kommunikation der BTnodes untereinander optimiert werden: Durch synchrones Ein- und Ausschalten der Radiomodule sind große Energieeinsparungen möglich, welche die Laufzeit der BTnodes mit einem Batteriesatz verlängern.

In einer vorangehenden Arbeit ist die Uhrensynchronisation auf BTnodes implementiert worden. In dieser Arbeit soll die Synchronisation mit einer konkreten Anwendung so verschmolzen werden, dass die Anwendung von der Synchronisation profitieren kann. Anschliessend sollen die Auswirkungen der Synchronisation (Zeit, Energie) untersucht werden.

Aufgabenstellung

1. Lesen Sie das „Merkblatt für Studenten und Betreuer“ [12] und machen Sie sich mit dem Bewertungsschema vertraut. So wissen Sie gleich von Anfang an, worauf es ankommt, um eine gute Note zu erzielen.
2. Erstellen Sie einen Projektplan und legen Sie Meilensteine fest. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
3. Verschaffen Sie sich einen Überblick über Uhrensynchronisation [11] mit Schwerpunkt auf intervallbasierte Algorithmen [10, 5].
4. Machen Sie sich mit dem Aufbau und der Funktionsweise der BTnodes [4, 1, 6] sowie mit der vorangehenden Arbeit [13] vertraut.

5. Machen Sie sich mit der BTnut-Software [7] vertraut; lernen Sie die JAWS-Applikation [9] als Beispielanwendung kennen.
6. Machen Sie sich mit der BTnut-Entwicklungsumgebung vertraut und implementieren Sie den Synchronisationsalgorithmus „IM“ aus [5]. Identifizieren Sie hierzu die ideale Systemebene in BTnut, auf der Synchronisationsalgorithmen implementiert werden können, und verwenden Sie zunächst das geeignetere der beiden Radios der BTnodes.
7. Implementieren Sie den Synchronisationsalgorithmus „BP-ISA“ aus [5]. Verwenden Sie auch das andere Radio der BTnodes.
8. Testen Sie die Funktion Ihrer Implementation und analysieren Sie die Abhängigkeit der Synchronisationsqualität von Parametern wie Kommunikationshäufigkeit und Prozessorlast. Vergleichen Sie die beiden implementierten Algorithmen und die beiden Radios.
9. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem abschliessenden Vortrag sowie mit einem Bericht. Die Qualität der Dokumentation fliesst in die Bewertung der Arbeit ein.

Durchführung der Arbeit

Allgemeines

- Wichtig: Dokumentieren Sie Ihre Arbeit vom ersten Tag an. Halten Sie sich hierbei an die Richtlinien in [12]. Vertrauen Sie nicht auf Ihr Gedächtnis.
- Der Verlauf der Arbeit soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme können Änderungen am Projektplan erforderlich machen. Auch diese sollen dokumentiert werden.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihrem Betreuer (persönlich, per E-Mail oder telefonisch). Sie können sich jederzeit mit Fragen an Ihren Betreuer wenden. Überlegen Sie sich aber Ihre Fragen und mögliche Lösungswege bzw. nächste Schritte in Ihrer Arbeit im Voraus, und diskutieren Sie diese dann mit dem Betreuer. Zeigen Sie Initiative [12].
- Sie sollten Ihr Projekt zu Beginn der Arbeit in einem Kurzvortrag (5 Minuten, 2-3 Folien) vorstellen. Hier werden natürlich noch keine Resultate erwartet. Am Ende der Arbeit sollen Sie Ihre Resultate im Rahmen eines 15- bis 20-minütigen Vortrags präsentieren.

Abgabe

- Geben Sie vier unterschriebene Exemplare des Berichts spätestens am festgelegten Abgabedatum beim Betreuer oder nötigenfalls im Institutssekretariat (ETZ G88) ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Dateien, Verzeichnisstrukturen usw. bestehen bleiben. Programmcode und Filestruktur sollen ausreichend dokumentiert sein, so dass eine spätere Anschlussarbeit mit geringem Aufwand auf dem hinterlassenen Stand aufbauen kann. Geben Sie beim Betreuer eine Kopie des gesamten Verzeichnisbaums (inklusive allen programmierten Codes sowie der Quelldateien des Berichts) auf CD-ROM oder einem geeigneten Medium ab.

Literatur

[1] Jan Beutel, Matthias Dyer, Oliver Kasten, Matthias Ringwald, Frank Siegemund, and Lothar Thiele. Bluetooth smart nodes for ad-hoc networks. Technical report, `ftp://ftp.tik.ee.ethz.ch/pub/publications/TIK-Report167.pdf`.

[2] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping wireless sensor network applications with btnodes. In *1st European Workshop on Wireless Sensor Networks (EWSN 2004)*, number 2920 in Lecture Notes in Computer Science, Springer, pages 323–338, Berlin, January 2004.

[3] Jan Beutel, Oliver Kasten, and Matthias Ringwald. Btnodes - a distributed platform for sensor nodes. In *Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003)*. ACM Press, New York, Nov. 2003.

[4] Jan Beutel, Oliver Kasten, and Matthias Ringwald. Btnodes - applications and architecture compared. Technical report, `ftp://ftp.tik.ee.ethz.ch/pub/people/beutel/BKR2003.pdf`, 2003.

[5] Philipp Blum, Lennart Meier, and Lothar Thiele. Improved interval-based clock synchronization in sensor networks. In *Third International Symposium on Information Processing in Sensor Networks*, pages 349–358, Berkeley, California, USA, April 2004.

[6] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. `http://www.btnode.ethz.ch`.

[7] BTnut system software. `http://btnode.ethz.ch/support/btnut_api/`, 2004.

[8] Jaap C. Haartsen. The Bluetooth Radio System. *IEEE Personal Communications*, Febr. 2000.

[9] JAWS application documentation. `http://www.btnode.ethz.ch/projects/jaws/jaws_api/`, 2004.

[10] Lennart Meier, Philipp Blum, and Lothar Thiele. Internal synchronization of drift-constraint clocks in ad-hoc sensor networks. In *Fifth ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 90–97, Tokyo, Japan, May 2004.

[11] Kay Römer, Philipp Blum, and Lennart Meier. *Sensor Networks*, chapter Time Synchronization and Calibration in Wireless Sensor Networks. Wiley & Sons, New York, 2005.

[12] Eckart Zitzler and Lennart Meier. Studien- und Diplomarbeiten, Merkblatt für Studenten und Betreuer. ETH Zürich, TIK, Mar. 2004.

[13] Boris Zweimüller. Uhrensynchronisation in einem Ad-Hoc-Netz von BTnodes. Studienarbeit SA-2004-06, WS 2003/2004, 2004.

Zürich, den 18. Oktober 2004

Felix Michel Philippe Wüger Lennart Meier

Angewandte Uhrensynchronisation auf BTnodes

Felix Michel Philippe Wüger

Semesterarbeit 2005-02
Betreuer: Lennart Meier

Institut für Technische Informatik und Kommunikationsnetze
D-ITET, ETH Zürich
Prof. Dr. Lothar Thiele

Wintersemester 2004/2005

Zusammenfassung

BTnodes sind kleine, programmierbare Netzknoten, die im Wesentlichen aus einem Mikrokontroller und zwei Radiomodulen bestehen, mit welchen drahtlose Kommunikation unter anderem nach dem Bluetooth-Standard möglich ist. Aus mehreren BTnodes kann beispielsweise ein Netz kommunizierender Sensorknoten aufgebaut werden. Die Synchronisation der BTnode-Uhren erlaubt in diesem Fall das Zusammenfassen von verteilten Messergebnissen, sowie Kommunikation mit grossen Energieeinsparungen, da die Radiomodule nur zu vereinbarten Kommunikationszeitpunkten eingeschaltet sein müssen.

In der vorliegenden Arbeit wurde die Uhrensynchronisation auf den BTnodes implementiert. Die Funktionalität wurde direkt in das auf den BTnodes laufende Betriebssystem BTnut integriert, so dass darauf aufsetzende Anwendungen davon profitieren können. Der für die Synchronisation nötige Informationsaustausch zwischen den BTnodes findet über die von den Anwendungen verschickten Pakete statt. Diese enthalten neben den Anwendungs-Nutzdaten zusätzlich auch Informationen über die lokale BTnode-Zeit. Besondere Beachtung musste der Nachrichtenverzögerung geschenkt werden, welche relativ hoch und starken Schwankungen unterlegen ist.

Abschliessend wurde die Implementierung getestet. Die erzielten Ergebnisse waren mit den Resultaten aus Simulationen vergleichbar.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 BTnodes

BTnodes sind kleine, programmierbare Netzknoten, welche als autonome Rechen- und Kommunikationsplattform im Bereich der mobilen Ad-hoc-Netze und in verteilten Sensornetzen dienen, wobei sie momentan hauptsächlich als Experimentier- und Demonstrationswerkzeug eingesetzt werden. Die wichtigsten Bestandteile sind Mikrocontroller sowie je ein Bluetooth- und ChipCon-Funkmodul. Als Betriebssystem dient BTnut, das auf Nut/OS [10], einem Open-Source-Betriebssystem für eingebettete Systeme, aufbaut. Die BTnodes wurden an der ETH Zürich am Institut für Technische Informatik und Kommunikationsnetze sowie am Institut für verteilte Systeme entwickelt und sind mittlerweile in der Revision 3.2 verfügbar, welche auch für die vorliegende Arbeit verwendet wurde.

1.2 Uhrensynchronisation

1.2.1 Motivation

In einem Netz von BTnodes ist die Synchronisation der Uhren der BTnodes wünschenswert. Einerseits ermöglicht sie das Zusammenfassen von verteilten Messergebnissen, indem z.B. aus den Zeitpunkten der verteilten Beobachtungen auf Kausalzusammenhänge globaler Vorgänge geschlossen wird, andererseits kann die Kommunikation der BTnodes untereinander optimiert werden: Verfügen zwei BTnodes über synchronisierte Uhren, so können sie Kommunikationszeitpunkte vereinbaren und ihre Radiomodule zwischen diesen Zeitpunkten ausschalten, wodurch grosse Energieeinsparungen möglich sind, welche die Laufzeit der BTnodes mit einem Batteriesatz verlängern.

1.2.2 Anforderungen in mobilen Ad-hoc-Netzen

Bei der Uhrensynchronisation sieht man sich mit zwei grundsätzlichen Problemen konfrontiert. Erstens verschlechtert sich die auf einem Knoten gespeicherte Zeitinformation andauernd, da die interne Hardware-Uhr nicht genau mit der richtigen Geschwindigkeit läuft. Zweitens besteht bei jeder Kommunikation mit einem anderen Knoten eine gewisse Unsicherheit bezüglich der Nachrichtenverzögerung, welche die ausgetauschte Zeitinformation weiter verschlechtert.

In mobilen Ad-hoc-Netzen werden zudem einige spezielle Anforderungen an die Uhrensynchronisation gestellt, so dass sich bereits bekannte Verfahren, wie zum Beispiel NTP [9], nicht umsetzen lassen:

- *Energieeffizienz* Die einzelnen Knoten sollen für die Uhrensynchronisation möglichst wenig Energie benötigen. Eine andauernde Kommunikation zur Aufrechterhaltung der Synchronisation ist deshalb nicht möglich. Das Verfahren muss vor allem auch bei sporadischem Nachrichtenaustausch funktionieren.[1]

- *Robustheit / Dynamische Netzwerke* Die Topologie eines Sensornetzes variiert stark: die einzelnen Knoten sind mobil, können aufgrund aufgebrauchter Batterien plötzlich nicht mehr erreichbar sein oder werden neu in ein Netz eingefügt. Es ist daher nicht garantiert, dass zwischen zwei Knoten eine stabile Verbindung mit konstanter Nachrichtenverzögerung aufgebaut werden kann, wie es z.B. NTP voraussetzt.

- *Autokonfiguration* Die einzelnen Knoten sollen nicht konfiguriert werden müssen.[2] Da die betrachteten Netzwerke aus einer Vielzahl von Knoten bestehen können, wäre der resultierende Aufwand nicht zu bewältigen.

  Näheres zu den erwähnten Punkten ist in [4] zu finden.

1.3 Zielsetzung

In dieser Arbeit sollen zwei gegebene Algorithmen zur Uhrensynchronisation auf dem BTnode implementiert und verglichen werden. Die entsprechenden Funktionen sollen direkt in BTnut integriert werden, so dass Anwendungen

[1] Im Streben nach Energieeffizienz wird dafür meist in Kauf genommen, dass die Knoten nicht permanent mit maximaler Genauigkeit synchronisiert sind.

[2] Zum Beispiel muss bei NTP die Adresse eines Zeitservers konfiguriert werden.

die Synchronisation als Dienst des Betriebssystems nutzen können. Typische C-Funktionen wie `gettimeofday()` sollen damit auch auf dem BTnode verfügbar werden. Die Implementierung soll möglichst energieeffizient realisiert werden.

Ein zweites Ziel ist der Vergleich der beiden Algorithmen IM und BP-ISA in ihrer tatsächlichen Implementierung untereinander und im Vergleich zu den Simulationen aus [1].

1.4 Übersicht

In Kapitel 2 werden das verwendete Uhrenmodell und die zwei Algorithmen IM und BP-ISA zur Uhrensynchronisation vorgestellt. Danach wird in Kapitel 3 näher auf die Implementierung dieser Algorithmen eingegangen. In den darauffolgenden Kapiteln werden Tests und Messungen beschrieben, um schliesslich gewonnene Erkenntnisse und Resultate zu diskutieren.

Kapitel 2

Algorithmen

2.1 Uhrenmodell

Eine Uhr wird auf einem Knoten durch einen Zähler repräsentiert, welcher – getaktet durch einen Oszillator der Frequenz f – jeweils nach einer gewissen Zeitspanne inkrementiert wird. Man bezeichnet die Zeit auf dem Knoten zum Zeitpunkt t als *lokale Zeit* $h(t)$. Idealerweise wäre die Zählrate $f(t) = dh(t)/dt$ zu allen Zeiten 1. Aufgrund von Variationen der Batteriespannung, Temperaturschwankungen usw. ist dies jedoch nicht der Fall. Die Abweichung der Zählrate von der idealen Rate wird als *Drift* ρ bezeichnet: $\rho(t) = f(t) - 1$. Oft kann dieser als begrenzt angenommen werden; die Hersteller von Oszillatoren und Resonatoren geben solche Schranken in den Datenblättern ihrer Bauteile an. Bei Uhren, für deren Drift Schranken angegeben werden können, spricht man vom "bounded-drift"-Modell:

$$-\rho_{max} \leq \rho(t) \leq \rho_{max} \qquad \forall\, t \tag{2.1}$$

2.2 Intervallbasierte Synchronisation

Intervallbasierte Synchronisationsalgorithmen berechnen ein Intervall, in welchem sich die gültige Zeit *garantiert* befindet, anstatt eine Schätzung der aktuellen Zeit anzugeben. Die entsprechenden Intervallgrenzen werden als untere und obere Schranke bezeichnet. Ein Schrankenpaar $[T^l(h(t)), T^u(h(t))]$, das zum Zeitpunkt $h(t)$ Gültigkeit hat und im Folgenden als $\vec{T}(h(t))$ bezeichnet sei, heisst *garantiert*, falls gilt:

$$T^l(h(t)) \leq t \leq T^u(h(t)) \qquad \forall\, t \tag{2.2}$$

Das Ziel jedes einzelnen Knotens besteht darin, die Grösse dieses Intervalls – die Unsicherheit über die Zeit – zu minimieren. Vorteile dieses Verfahrens:

- Die Kombination zweier Intervalle von verschiedenen Knoten ist eindeutig und optimal. Im Gegensatz dazu bedingt die Kombination zweier Zeitschätzungen zusätzliche Information über die Qualität der Schätzungen.
- Die reale Zeit ist garantiert zwischen der oberen und unteren Schranke. Synchronisierte Knoten können sich bei einer zu einem bestimmten Zeitpunkt verabredeten Kommunikation nie verpassen, wenn sie das Funkmodul einschalten, solange sich der Zeitpunkt zwischen den Schranken befindet.
- Die Knoten wissen immer über die aktuelle Zeitunsicherheit Bescheid.

Für die vorliegende Arbeit wurden die Algorithmen IM und BP-ISA betrachtet, welche nachfolgend beschrieben werden. Eine formalere Beschreibung ist in [1] zu finden.

2.2.1 IM

Der Algorithmus IM wurde erstmals von Keith Marzullo und Susan Owicki in [2] vorgeschlagen. Bezogen auf die Zeitunsicherheit ist dieser Algorithmus *worst-case-optimal*. Jeder Knoten speichert eine obere (T_M^u) und untere (T_M^l) Schranke der Zeit, sowie den Zeitpunkt h_M, zu dem diese Schranken berechnet wurden. Soll eine Nachricht mit Zeitinformationen zur Zeit h verschickt werden, so werden daraus die aktuellen Zeitschranken berechnet:

$$T^l(h) = T_M^l + \frac{\Delta h}{1+\rho_{max}} \qquad T^u(h) = T_M^u + \frac{\Delta h}{1-\rho_{max},} \tag{2.3}$$

wobei $\Delta h = h - h_M$ gilt.

Erhält ein Knoten von einem anderen Knoten Zeitschranken, so werden diese mit den eigenen kombiniert und zusammen mit der lokalen Zeit abgespeichert:

$$T_M^l = \max(T_{current}^l, T_{received}^l) \qquad T_M^u = \min(T_{current}^u, T_{received}^u) \tag{2.4}$$

$$h_M = h$$

Zu Beginn werden T_M^l und T_M^u mit $-\infty$ bzw. $+\infty$, h_M mit 0 initialisiert.

2.2.2 BP-ISA

Wie IM ist auch der "Back-Path Interval Synchronization Algorithm" (BP-ISA) *worst-case-optimal*, erzielt jedoch in non-worst-case-Situationen bessere

Resultate. Ein Knoten speichert nun für jeden erreichbaren Nachbarknoten N_i die Schranken $\vec{T}_M[N_i]$ sowie die Zeitpunkte $h_M[N_i]$ der letzten Kommunikation mit dem Knoten. Beim Versenden einer Nachricht an Knoten N_i werden neben den aktuellen Schranken $\vec{T}_{current}$ auch die entsprechenden vergangenen Schranken $\vec{T}_M[N_i]$ mitgeschickt. Ein empfangender Knoten benutzt beide Intervalle, um die bei ihm gespeicherten Schranken zu verbessern. Dazu muss er auch Schranken für ein negatives Δh berechnen können, so dass die Berechnungsvorschrift der Schranken für negative Lokalzeitintervalle Δh ergänzt werden muss:

$\Delta h \geq 0$:

$$T^l(h) = T^l_M + \frac{\Delta h}{1+\rho_{max}} \qquad T^u(h) = T^u_M + \frac{\Delta h}{1-\rho_{max}}$$

$\Delta h < 0$:

$$T^l(h) = T^l_M + \frac{\Delta h}{1-\rho_{max}} \qquad T^u(h) = T^u_M + \frac{\Delta h}{1+\rho_{max}}$$

Die untere Schranke beispielsweise wird nun beim "Vorwärtsrechnen" ($\Delta h \geq 0$) wie bei IM mit minimaler Geschwindigkeit erhöht, beim "Rückwärtsrechnen" ($\Delta h < 0$) – was bei IM nicht vorkommt – aber mit maximaler Geschwindigkeit erniedrigt, damit die Schranken ihre Gültigkeit garantiert behalten.

Die aktuellen Schranken werden immer aus den beim letzten Nachrichtenaustausch hervorgegangenen Schranken berechnet, wofür der Knoten, mit dem das letzte Mal kommuniziert wurde, mit einer Variable N_M referenziert werden muss.

Die Initialisierung erfolgt wie bei IM, bloss muss sie bei BP-ISA für die Schranken $\vec{T}[N_i]$ und Zeitpunkte $h_M[N_i]$ *aller* Nachbarn geschehen.

Der Algorithmus soll im Folgenden kurz *anschaulich* in Pseudo-Code dargestellt werden. Für eine vollständige und formalere Darstellung sei auf [1] verwiesen.

Globale Variablen:

- $\vec{T}[N_i]$ bezeichne die Schranken $[T^l[N_i], T^u[N_i]]$, die bei der letzten Kommunikation mit dem Nachbarknoten N_i gespeichert worden sind,
- $h_{lastUpdate}[N_i]$ bezeichne den Zeitpunkt der letzten Kommunikation mit N_i, und
- $N_{mostRecent}$ identifiziere den Nachbarknoten, mit dem zuletzt kommuniziert wurde.
- $h(t)$ gebe die lokale Zeit an.

Folgende Prozeduren werden benötigt:

```
updateBounds{in: T⃗_toBeUpdated, Δh / out: T⃗_updated}
    if Δh ≥ 0
        T⃗_updated = T⃗_toBeUpdated + [ Δh/(1−ρ̂) ; Δh/(1+ρ̂) ]
    else
        T⃗_updated = T⃗_toBeUpdated + [ Δh/(1+ρ̂) ; Δh/(1−ρ̂) ]
    end

intersectBounds{in: T⃗_comparand, T⃗_comparator / out: T⃗_intersected}
    T⃗_intersected = [ min(T⃗^u_comparand, T⃗^u_comparator) ; max(T⃗^l_comparand, T⃗^l_comparator) ]
```

```
currentBounds{in: - / out: T⃗_current}
    Δh = h − h_lastUpdate[N_mostRecent]
    T⃗_current = updateBounds(T⃗[N_mostRecent], Δh)

updateMemory{in: T⃗_updateWith, h_updateWith / out: -}
    foreach N_i
        Δh = h_lastUpdate[N_i] − h_updateWith
        T⃗_comparable = updateBounds(T⃗_updateWith, Δh)
        T⃗[N_i] = intersectBounds(T⃗[N_i], T⃗_comparable)
    end
```

```
processMessage{in: T⃗_new, T⃗_previous, N_sender / out: -}
    updateMemory(T⃗_previous, h_lastUpdate[N_sender])
    T⃗[N_sender] = intersectBounds(currentBounds(), T⃗_new)
    h_lastUpdate[N_sender] = h
    N_mostRecent = N_sender
    updateMemory(T⃗[N_sender], h)

generateMessage{in: N_destination / out: T⃗_new, T⃗_previous}
    T⃗_new = currentBounds()
    T⃗_previous = T⃗[N_destination]
```

Die ersten zwei Prozeduren sind (abgesehen vom Fall negativer Δh in `updateBounds`) mit denjenigen des Algorithmus IM identisch; die letzten beiden Prozeduren sind diejenigen, die bei einem Sende- (`generateMessage`) oder Empfangsereignis (`processMessage`) ausgeführt werden. Die entschei-

dende Prozedur ist `updateMemory`, in der zweimal – zuerst mit $\vec{T}_{previous}$, dann mit dem neu berechneten $\vec{T}_{current}$ – versucht wird, alle gespeicherten Schranken zu verbessern. Zu diesem Zweck werden die Schranken, mit denen `processMessage` aufgerufen wird (also $\vec{T}_{updateWith}$ innerhalb der Routine `updateMemory`) mittels `updateBounds` auf den Zeitpunkt $h_{lastUpdate}[N_i]$, der den gespeicherten Schranken $\vec{T}[N_i]$ zugeordnet ist, vor- oder zurückgerechnet, um die beiden Intervalle dann schneiden zu können. Dadurch kann die Unsicherheit im allgemeinen Fall verkleinert werden. Man kann dies an folgendem Beispiel nachvollziehen:

- Zwei Knoten N_1, N_2 (mit den jeweiligen lokalen Uhren $h_1(t), h_2(t)$) kommunizieren zum Zeitpunkt t_0.
- Zu einem späteren Zeitpunkt t_1 kann der Knoten N_2 seine Schranken durch Kommunikation mit einem dritten Knoten verbessern. Auch die in $\vec{T}[N_1]$ gespeicherten Schranken, die vom Kommunikationsereignis von t_0 stammen, werden verbessert.
- Wenn nun zur Zeit t_2 Knoten N_1 und N_2 wieder kommunizieren, so sendet N_2 auch die gespeicherten und durch die Kommunikation zur Zeit t_1 verbesserten Schranken als $\vec{T}_{previous}$ mit. Mit dieser zusätzlichen Information können N_1 und N_2 die Unsicherheit ihrer Schranken weiter verkleinern. Da sozusagen "im Nachhinein" Information durch die Kommunikation beim Ereignis zur Zeit t_0 geflossen ist (die damals nicht fliessen konnte, weil sie ja erst bei t_1 gewonnen wurde), ist nun der *backpath* von t_1 über das Kommunikationsereignis bei t_0 bis zu demjenigen von t_2 berechnet worden.

Die Verbesserung von BP-ISA gegenüber IM kann durch Darstellung des Problems als Graph genauer erklärt werden, wie es in [1] gemacht wird.

Kapitel 3

Implementierung

3.1 Übersicht

In diesem Kapitel werden zuerst die allgemeinen Überlegungen zur Implementierung geschildert. Es wird dabei unterschieden zwischen Überlegungen, die durch die in der Aufgabenstellung definierten Ziele oder durch Anforderungen, die für Algorithmen in verteilten Sensornetzen allgemein gelten, geleitet wurden und Entwurfsentscheiden, die wir gefällt haben, die aber auch anders möglich gewesen wären. Weiter unterscheiden wir Änderungen und Erweiterungen der gegebenen Algorithmen, die bei der Implementierung nötig wurden.

Anschliessend wird näher auf den Quellcode eingegangen. Neu eingeführte Datentypen, die wichtigsten globalen Variablen und die Synchronisationsfunktionen werden kommentiert. Der prinzipielle Ablauf beim Senden und Empfangen von Paketen soll den ganzen Vorgang illustrieren und die einzelnen Funktionen in einen Zusammenhang stellen. Weitere Implementierungsmöglichkeiten, welche den Rahmen der Arbeit gesprengt hätten, werden im letzten Abschnitt konzeptuell beschrieben.

3.2 Konzept - Grundlegende Überlegungen

Die Uhrensynchronisation sollte nicht selbst eine Anwendung auf den BTnodes sein, vielmehr sollten andere Applikationen von ihr profitieren können. Die Uhrensynchronisation sollte deshalb direkt in BTnut integriert werden und so transparent den auf dem BTnode laufenden Anwendungen zur Verfügung stehen. Überdies sollte den in 1.2.2 beschriebenen speziellen Anforderungen Rechnung getragen werden.

3.2.1 Effizienz, Robustheit, Autokonfiguration

Da Effizienz nicht nur bezüglich *einer* Ressource, sondern hinsichtlich Speicherbedarf, Rechenaufwand und Energieverbrauch gefordert ist, war ein Abwägen zwischen den drei Kriterien nötig. Wir haben tendenziell einen höheren Speicherbedarf in Kauf genommen, um den Rechenaufwand gering zu halten, da letzterer ja auch den Energieverbrauch massgeblich mitbestimmt. Auch der Anspruch, einen transparenten Dienst zu entwerfen, implizierte sparsame Verwendung von Prozessorzeit. Recheneffizienz wurde zudem mit dem Verzicht auf Divisionen und Modulo-Operationen in den Berechnungsfunktionen angestrebt. Die beiden anderen Punkte (Robustheit und Fähigkeit zur Autokonfiguration) verlangten weitgehende Unabhängigkeit von spezifischen Netztopologien. Auch auf denkbare Overlay-Strukturen[1] wurde verzichtet, da dies ein weit komplexeres Protokoll bedingt hätte, welches wiederum einen Mehraufwand zur Folge gehabt hätte. Unsere Implementierung benötigt denn auch keine spezielle Ordnung oder verschiedene Rollen der Knoten. Auch die sogenannten Ankerknoten, über die die Referenzzeit ins Netz eingespiesen wird und die deshalb strenggenommen eine Sonderrolle einnehmen, verbergen ihre unterschiedliche Funktionalität sowohl den Nachbarknoten als auch den auf ihnen laufenden Applikationen gegenüber.

3.2.2 Entwurfsentscheide

Obwohl einige Entscheide bereits vorweggenommen worden sind, werden hier nochmals unsere Entscheide bei Problemstellungen der Implementierung aufgelistet und mögliche alternative Wege angedeutet. Genaueres zur tatsächlichen Implementierung ist in Kapitel 3.3 zu finden.

- Die Synchronisation soll implizit geschehen: Es soll keine zusätzliche Kommunikation zur Synchronisation erfolgen, vielmehr wird die Zeitinformation in den Datenpaketen, die ohnehin versandt werden, "huckepack" (*piggy-back*) mitgeschickt. Das Bestreben, zwecks Energieeffizienz die Kommunikation minimal zu halten, legte dies nahe, da so praktisch kein Overhead entsteht. Eine Alternative wäre gewesen, explizit zu synchronisieren, welche wegen des eben erwähnten Overheads und des Wunsches, Synchronisation als transparenten Dienst zu verwirklichen, ausschied. Eine dritte Möglichkeit ist, adaptiv zwischen impliziter und expliziter Synchronisation zu wechseln. So könnte zum Beispiel beim Überschreiten einer gewissen Unsicherheitsschwelle explizit Zeit-

[1] Naheliegend wäre zum Beispiel ein Baum minimaler Tiefe mit dem Ankerknoten als Wurzel, wie es ähnlich auch in [9] geschieht.

information angefordert werden. Diese Variante ist in Abschnitt 3.5 auf Seite 38 beschrieben.

- Das Protokoll soll so einfach wie möglich gehalten werden. Die Schranken werden immer mitgesandt und sind in der Payload enthalten, zusammen mit Flags, welche die Gültigkeit der Schranken angeben.

- Auch das Zeitformat wird einfach gewählt. Obere und untere Schranken werden als Millisekundenzeitstempel versandt und verarbeitet. Sowohl zum Protokoll als auch zum Zeitformat findet sich Genaueres in Abschnitt 3.3 und Alternativen in Abschnitt 3.5.

- Als Radiomodul haben wir das Bluetooth-Modul ausgewählt. Der Entscheid, implizit zu synchronisieren legte dies nahe, da für Bluetooth bereits ein Protokollstapel definiert und in BTnut verwirklicht ist (BTstack, vgl. [3]). Für das ChipCon-Radio fehlen zur Zeit noch höhere Layer; dafür wäre der direktere Zugriff auf die MAC-Schicht eventuell günstiger, um die Unsicherheit über die Nachrichtenverzögerung klein zu halten.

- Die Synchronisationsfunktionalität soll auf der tiefsten im BTstack erreichbaren Ebene implementiert werden. Zwar liesse sich mit dem Eingreifen in die Hardwaretreiber die Unsicherheit über die Verzögerung der Pakete auf dem BTnode vielleicht weiter verringern. Das Umschreiben der Treiber, die in Nut/OS "generisch"[2] entworfen sind, wirft aber wieder neue Probleme auf. Auch ein Eingriff in die Firmware des Bluetooth-Moduls lag jenseits der zeitlichen Möglichkeiten.

- Ankerknoten sollen ihre von Sensorknoten unterschiedliche Gewinnung der Zeitinformation (nämlich Bezug bestmöglicher Schranken vom PC anstatt Berechnung aus gespeicherten Schranken) sowohl gegenüber den Nachbarknoten als auch gegenüber den auf ihnen laufenden Anwendungen verbergen. Die Schnittstellen sollen sich möglichst nicht von denjenigen eines Sensorknotens unterscheiden, damit Anwendungen nicht in mehrfachen Versionen entwickelt werden müssen.

3.2.3 Anpassung des Algorithmus

In der Beschreibung des Algorithmus BP-ISA in [1] ist von "communication events" die Rede, einem "event c which occurs at both nodes at real time

[2] Vgl. zum Beispiel `usartavr.c`: Die relevanten Funktionen werden sowohl von der `USART0` als auch von der `USART1` verwendet.

t_c."[3] Weiter heisst es: "The nodes simultaneously acquire mutual knowledge about their time bounds."[4] Diese Annahme – gleichzeitiges gegenseitiges Versenden der Schranken zweier BTnodes – wird sich in der realen Welt nicht umsetzen lassen. Die Forderung nach echter Gleichzeitigkeit ist in verteilten Netzen ohnehin schwer zu erfüllen; eine geplante gleichzeitige gegenseitige Kommunikation ist in der Praxis zudem erst möglich, wenn die Uhren der kommunizierenden Knoten bereits synchron sind. Nicht zuletzt scheiterte das Unterfangen in der praktischen Implementierung, da zwar die UART des Mikrokontrollers potenziell voll-duplex-fähig wäre (vgl. [6]), die Treiber dies aber nicht unterstützen.

Es stellte sich die Frage, inwiefern Gleichzeitigkeit und Gegenseitigkeit der Kommunikation eine Bedingung für die Korrektheit der Algorithmen ist. Gleichzeitigkeit scheint das geringere Problem darzustellen, da sie auch bei nicht-gegenseitiger Kommunikation gegeben sein kann: Das Kommunikationsereignis wird in zwei Einzelereignisse – ein Sende- und ein Empfangsereignis – aufgespaltet, die in der Idealisierung ohne Nachrichtenverzögerung auch dann noch gleichzeitig sind, wenn nur in eine Richtung eine Nachricht fliesst. Das Problem wird also auf jenes der Gegenseitigkeit zurückgeführt. Ohne einen formalen Beweis zu geben lässt sich mit folgender Überlegung leicht zeigen, dass auch Gegenseitigkeit keine Bedingung für die Korrektheit, sondern höchstens ein Faktor der erzielbaren Synchronisations-Qualität ist. Wir betrachten den Fall eines gegenseitigen Kommunikationsereignisses zwischen N_1 und N_2, bei dem alle vier Schranken ($\vec{T}^{N_1}_{new}$, $\vec{T}^{N_1}_{previous}$), die N_1 versendet, zu allen möglichen Referenzzeitpunkten eine grössere Unsicherheit haben als alle Schranken, die N_2 besitzt; $\vec{T}^{N_1}_{new}$ wäre also zum Beispiel $(-\infty, \infty)$. Dann haben alle Prozeduren, die beim Empfangen der Nachricht auf N_2 aufgerufen werden, keinen Einfluss auf das Wissen von N_2 über die Zeit. Aus Sicht der Synchronisation ist dies dem Fall gleichzusetzen, in dem N_2 *überhaupt keine* Schranken erhalten hätte, in dem die Kommunikation also einseitig von N_2 nach N_1 verlaufen wäre. Da solche Schranken für BP-ISA nicht verboten sind und sich dieser Fall aber in denjenigen einseitiger Kommunikation überführen lässt, sollte Gegenseitigkeit auch keine Bedingung für korrektes Funktionieren von BP-ISA sein.

Der Schluss, dass der Algorithmus BP-ISA auch ohne gleichzeitige gegenseitige Kommunikation funktioniere, lässt sich überdies anschaulich zeigen, indem der einfache Fall, der in [1] in *Abbildung 3* skizziert und auch in dieser

[3] In [1] unter "Scenarios and Executions". *Real time* wird hier im Sinne von *tatsächlicher Zeit* (oder Referenzzeit) verwendet und nicht in dem Sinn, wie es in "Echtzeitverarbeitung" und Ähnlichem auftritt.

[4] ebenda

Arbeit unter 2.2.2 als Beispiel dient, für alle vier Fälle[5] durchgerechnet wird.

Es ist jedoch eine kleine Änderung am Algorithmus BP-ISA vonnöten, um die Konsistenz der zu den gespeicherten Schranken $\vec{T}_M[N_i]$ gehörigen Zeitpunkte $h_M[N_i]$ auf dem Sende- und denjenigen auf dem Empfangsknoten aufrecht zu erhalten. In der Art und mit den Variablen des Pseudo-Codes von Seite 8 sähe diese Ergänzung so aus:

Die Prozedur `generateMessage` wird abgeändert zu:

```
generateMessage{in: N_destination / out: T_new, T_previous}
    T_new = currentBounds()
    T_previous = T[N_destination]

    Δh = h − h_lastUpdate[N_destination]
    T[N_destination] = updateBounds(T[N_destination], Δh)
    h_lastUpdate[N_destination] = h
```

Das Problem der Gegenseitigkeit ist damit gelöst. Gleichzeitigkeit kann in der Praxis nicht erreicht werden, aber bei Kenntnis der Nachrichtenverzögerung (oder zumindest Schranken für sie) lässt sich auch dabei Korrektheit erzielen, allerdings zum Preis einer zusätzlichen Unsicherheit und somit schlechteren Synchronisationsqualität.

3.2.4 Zusätzliche Konzepte

Es stellte sich heraus, dass die Unsicherheit über die Nachrichtenverzögerung ein zentrales Problem bei der Implementierung ist. Während wir zuerst versuchten, der Unsicherheit mit statistischen Überlegungen Herr zu werden, sahen wir später ein, dass eine Messung der Nachrichtenverzögerung zur Laufzeit notwendig ist. Dafür mussten die nötigen Datenstrukturen geschaffen werden, da Sende- und Empfangszeitpunkte für jeden erreichbaren Nachbarn gespeichert werden müssen. Das genaue Vorgehen wird in Abschnitt 3.3.7 erläutert.

[5] Die Kommunikation zu den Zeiten t_0 und t_2 kann in jeweils eine der zwei Richtungen geschehen; bei derjenigen zur Zeit t_1 sollte N_2 der Empfänger der Information sein, wenn die Annahme weiterhin gelten soll, dass N_2 damit seine Schranken sicher verbessert.

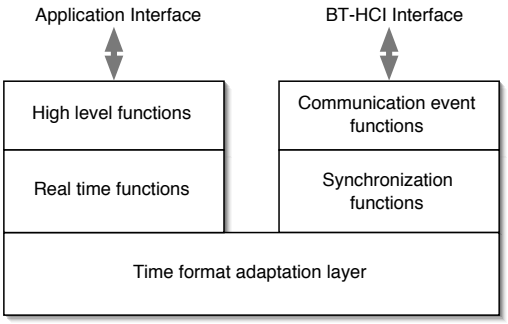

Abbildung 3.1: Einfaches Schichtenmodell.

3.2.5 Schichtenmodell

In der Hoffnung, die Implementierung übersichtlicher und modularer zu machen, wurde ein grobes Schichtenmodell (Abbildung 3.1) als Orientierung definiert. Es ist keinesfalls ein echtes Schichtenmodell mit genau definierten Schnittstellen und Anforderungen, wie es etwa die Protokollstapel in Kommunikationsprotokollen sind. Auch ist die Gliederung nicht identisch mit derjenigen in Abbildung 3.5 auf Seite 31, welche das genauere Zusammenspiel der implementierten Funktionen wiedergibt. Es soll aber zeigen, welche Gruppen von Funktionen konzeptuell zusammengehören, wo allfällige Ergänzungen anschliessen sollten und welche Layer bei alternativen Implementierungen ersetzt werden könnten, ohne dass andere davon betroffen wären. Die Layer und einige beispielhafte Funktionen daraus werden nun kurz beschrieben:

- Die höheren beiden Schichten links im Bild enthalten Funktionen, die Informationen über die "Echtzeit", also die Referenz- oder tatsächliche Zeit, an Applikationen liefern (z.B. `rt_is_valid()`) beziehungsweise die Funktionen, welche die dafür nötigen Operationen auf den aktuellen Schranken ausführen (z.B. `get_updated_bounds()`). Die Funktionen der obersten Schicht sind mit dem Präfix "`rt`" versehen.

- Die beiden obersten Schichten rechts im Bild stellen das Interface zu den Bluetooth-Funktionen dar und tragen das Präfix "`sync`". Sie enthalten die beim Empfang und beim Senden aufgerufenen Funktionen (z.B. `sync_generate_timestamp()`) und jene, welche die tatsächliche Synchronisation ausführen (z.B. `sync_sync_bounds()`).

- Der unterste Layer vermittelt zwischen den verschiedenen Darstellungsformen der Zeitinformation, insbesondere 6-Byte-Character-String- und

`u_longlong`-Werten. Hierzu zählen zum Beispiel Funktionen wie `sync_disassemble_timestamp()`; auch der Datentyp `syncbounds`, der sozusagen als "lingua franca" zwischen den Schichten dient, lässt sich dieser Schicht zuordnen. Diese Schicht könnte zum Beispiel dereinst neu implementiert werden, wenn ein anderes Zahlenformat beim Übertragen gewünscht wird, ohne dass die Funktionen der anderen Layer verändert werden müssten.

3.3 Implementierung

Die wichtigsten Punkte werden nun noch einmal unter dem Aspekt der tatsächlichen Implementierung genauer beleuchtet.

3.3.1 Implizite Uhrensynchronisation

Grundsätzlich geschieht der Austausch von Zeitinformationen implizit, d.h. die Informationen werden direkt in die von der Anwendung versandten Datenpakete eingebettet. Dies ist sehr energieeffizient, da zur Synchronisation keine zusätzlichen Pakete versandt werden müssen, als ohnehin schon durch die Anwendung versendet werden.[6] Die implizite Synchronisation leidet jedoch darunter, dass keine Garantie abgegeben werden kann, ob und wie oft kommuniziert wird. Die Unsicherheit über die Zeit ist also abhängig von der Kommunikationsfrequenz der Anwendung.

Dieses Problem kann dadurch entschärft werden, dass den Anwendungen Funktionen zur Verfügung gestellt werden, welche die aktuellen Schranken zurückliefern (`rt_get_bounds()`) und mitteilen, ob der BTnode überhaupt im Besitz von gültigen Schranken ist (`rt_is_valid()`). Die Anwendung kann somit die Grösse der Unsicherheit über die Zeit selbst berechnen und so je nach dem verschiedene Aktionen in die Wege leiten. Beispielsweise könnte das Unsicherheitsintervall durch Kommunikation mit BTnodes, welche genauere Zeitinformationen besitzen, verkleinert werden. Eine andere Möglichkeit besteht darin, im Betriebssystem über den Synchronisationsfunktionen einen weiteren Layer einzuführen, welcher diese Funktionalität implementiert und den Anwendungen zur Verfügung stellt.

[6] Dies stimmt immerhin unter der Annahme sporadischer Kommunikation und geringer Datenmengen. In schlechteren Fällen (z.B. ein *Stream*, der die volle Bandbreite nutzt) kann die von der Synchronisation verursachte Herabsetzung der Payload (indirekt) sehr wohl die Paketzahl erhöhen, weil die Daten bei geringerer Payload auf $\lceil Payload/(Payload - Timestamp) \rceil$ mal mehr Pakete verteilt werden müssen.

3.3.2 Betriebssystemebene

Nach den in 3.2.2 gefällten Entwurfsentscheiden sollten die Synchronisationsfunktionen auf der tiefsten im BTstack erreichbaren Systemebene implementiert werden, um die Unsicherheit bezüglich Nachrichtenverzögerung klein zu halten und den Anwendungen einen transparenten Service anbieten zu können. Die tiefste im BTstack erreichbare Systemebene wird im Bluetooth-Standard als *Host-Controller Interface (HCI)* bezeichnet (vgl. [3], [7]). Über eine UART-Schnittstelle kann der Mikrocontroller mit dem Bluetooth-Modul kommunizieren und so Befehle und Daten senden und empfangen. Die implementierten Synchronisationsfunktionen greifen auch auf dieser Ebene ein. Wird ein Paket versendet, so wird der aktuelle Zeitstempel kurz vor dem Schreiben auf die UART-Schnittstelle generiert und in das Paket integriert. Beim Lesen eines Paketes von der Schnittstelle werden die Synchronisationsfunktionen umgehend benachrichtigt. Eine detailliertere Beschreibung der involvierten Funktionen und des Ablaufs ist in Abschnitt 3.4 zu finden.

3.3.3 Bluetooth

Im Bluetooth-Standard [7] sind 2 verschiedene Kommunikationstypen definiert. Einerseits gibt es *asynchronous connectionless (ACL)* Links, welche sich für den Austausch von Daten eignen. Andererseits gibt es *synchronous connection-oriented (SCO)* Links, welche vor allem für Audioanwendungen (z.B. Headsets) verwendet werden, da sie eine fest reservierte Bandbreite zur Verfügung stellen. Da SCO auf dem BTnode noch nicht vollständig implementiert ist und in Sensornetzwerken eine untergeordnete Rolle spielt, konzentrierten wir uns auf ACL-Links.

Auf HCI-Ebene sind verschiedene Pakettypen definiert, welche der Mikrokontroller mit dem Bluetooth-Modul austauschen kann. Mit `HCI_COMMAND_DATA_PACKET`-Paketen können Befehle ans Bluetooth-Modul geschickt werden. In die Gegenrichtung werden `HCI_EVENT_PACKET`-Pakete versandt. Für die eigentliche Datenübertragung werden `HCI_ACL_DATA_PACKET`- und `HCI_SCO_PACKET`-Pakete verwendet. Die Zeitstempel werden nur in die `HCI_ACL_DATA_PACKET`-Pakete integriert, da wir die Kommunikation mit SCO-Paketen – wie bereits erwähnt – nicht berücksichtigten.

Das Format eines solchen Paketes inklusive Zeitstempel (dessen Format in Abschnitt 3.3.4 noch erläutert wird) ist in Abbildung 3.2 dargestellt.

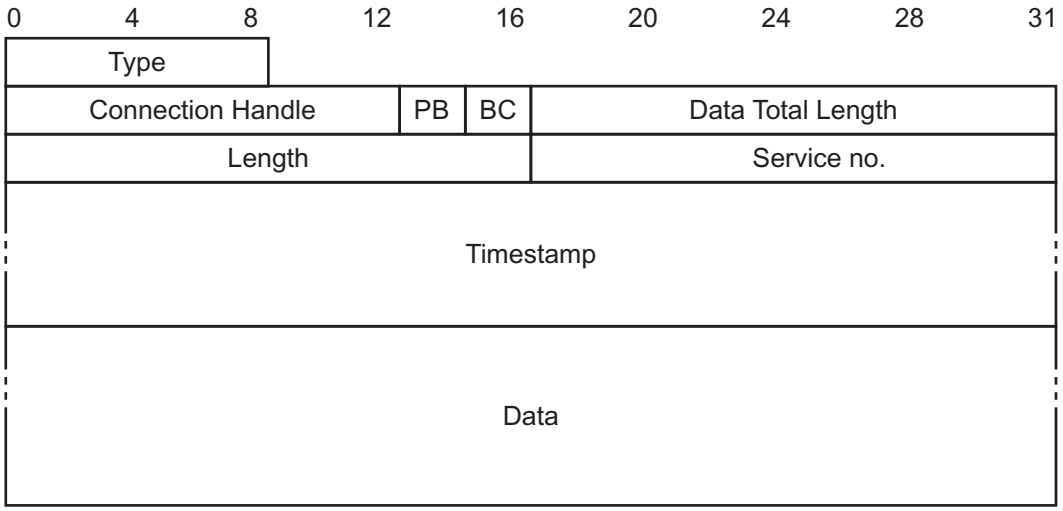

Abbildung 3.2: Ein um den "Timestamp" erweitertes ACL-Datenpaket.

3.3.4 Zeitformat

Für die Übertragung der Zeitinformation und deren Repräsentation auf dem BTnode muss ein geeignetes Zeitformat gewählt werden. Da diese Information in ein von der Anwendung verschicktes Paket eingebettet wird und sich somit die von der Anwendung beanspruchbare Payload reduziert, sollte die zusätzliche Datenmenge möglichst klein sein.

Erweitertes Unix-Zeitformat

Grundsätzlich soll die Zeit auf dem BTnode im Unix-Format angegeben werden können, welches die Anzahl vergangener Sekunden seit dem 1. Januar 1970 angibt. Dafür werden 31 Bit benötigt.[7] Zusätzlich soll die Granularität aber 1 Millisekunde betragen.[8] Es ist deshalb naheliegend, die Zeit durch die Anzahl vergangener *Milli*sekunden seit dem 1. Januar 1970 zu repräsentieren. Gegenüber den Sekunden erfordert dies weitere 10 Bit. Insgesamt sind für die volle Zeitrepräsentation also 41 Bit notwendig.

Es stellt sich jedoch die Frage, ob dieses Zeitformat auch für die interne Repräsentation der Zeit und für die Berechnungen in den Synchronisationsfunktionen geeignet ist, da in diesem Fall mit `long long` Werten (64

[7]Damit können Zeiten bis zum 19. Januar 2038 um 04:14:07 Uhr angegeben werden.

[8]Simulationen in [1] haben gezeigt, dass der Synchronisationsalgorithmus eine Genauigkeit im Millisekunden-Bereich erreichen kann.

Bit) gerechnet werden muss. Obwohl sich das mit alternativen Zeitformaten vermeiden liesse (vgl. das in Abschnitt 3.5 vorgeschlagene Präfixformat), haben wir uns entschieden, den Ablauf der Synchronisation einfach zu halten und somit auch mit `long long` Werten zu rechnen. Beim Versenden von Zeitinformationen wird somit das um Millisekundengranularität erweiterte Unix-Format beibehalten. Im Bewusstsein, dass Berechnungen mit `long long` Werten einen hohen Zeit- und Programmcodeaufwand mit sich bringen, haben wir versucht, die Anzahl dieser Operationen auf ein Minimum zu beschränken, wo immer möglich nur mit der relevanten Genauigkeit zu rechnen, Divisionen gänzlich zu vermeiden und beim Schreiben der betroffenen Funktionen auch ein Auge auf *code-size-efficiency* zu werfen. Zudem kann die Berechnungszeit abgeschätzt und bei der Synchronisation berücksichtigt werden. Als wesentlicher Nachteil[9] der `long long` Berechnungen bleibt somit nur, dass die maximale Datenrate bzw. Paketrate eventuell nicht ausgenutzt werden kann, was wir aber nicht als gravierend empfinden, da wir in Sensornetzen sowieso von geringen Datenraten ausgehen.

Einbettung in Payload

Für den Algorithmus IM muss jeweils eine untere und eine obere Schranke übertragen werden, was 82 Bit entspricht. Um noch weniger Platz im Paket zu belegen, könnte man auch nur eine Schranke und die Differenz zur anderen Schranke versenden. Dies setzt voraus, dass eine Annahme über die mögliche Intervallgrösse getroffen wird. Bei sporadischer Kommunikation kann diese je nach gewähltem maximalen Drift durchaus in die Grössenordnung von Minuten kommen. Für den Offset sind also um die 24 Bit[10] nötig, so dass diese Variante 65 Bit benötigen würde.

Die Implementierung kann vereinfacht werden, wenn die Länge der einzelnen Schranken bzw. des Offsets jeweils ein Vielfaches von 8 Bit ist, da die Daten *byteweise* über die UART-Schnittstelle zum Bluetooth-Modul geschickt werden. Eine Schranke nimmt somit – aufgerundet auf ein Vielfaches von 8 Bit – 48 Bit (6 Byte) in Anspruch. Die Übertragung zweier Schranken würde 12 Byte, das Versenden einer Schranke und eines Offsets $48 + 24 = 72$ Bit (9 Byte) benötigen.

Aus folgenden Gründen haben wir uns für die Verwendung der ersten Variante entschieden:

[9] Das ist leider nicht der einzige Nachteil: Im Lauf der Implementierung stellte sich heraus, dass der Compiler 64-Bit-Werte nur halbherzig unterstützt: So kann etwa `printf` den Formatstring `%lld` nicht interpretieren. Auch ist es unmöglich, in `asm`-Statements auf die höheren vier Bytes zuzugreifen.

[10] Auf ganze Byte gerundet.

- Die maximale Payload eines ACL-Datenpakets beträgt 117 Bytes. Eine Ersparnis von 3 Byte pro Paket rechtfertigt den höheren Implementierungsaufwand nicht, zumal man davon ausgehen kann, dass die in Sensornetzwerken ausgetauschten Datenmengen relativ gering sind.

- Das Verwenden von zwei vollen Schranken erlaubt beliebig grosse Unsicherheitsintervalle. Es muss keine Sonderbehandlung beim Erreichen der maximalen Intervallgrösse erfolgen, was beliebig sporadische Kommunikation erlaubt und die Implementierung weiter vereinfacht.

Aus den selben Gründen sind wir der Meinung, dass dieses simple Zeitformat auch für die Implementierung des Algorithmus BP-ISA geeignet ist, obwohl hier nun natürlich 4 volle Schranken übertragen werden und somit 24 Byte von der Payload beansprucht werden.

Die 2 respektive 4 Schranken (und die im Falle einer Messung der "round-trip time" zur Laufzeit zusätzlich nötige Information[11]), welche in ein Paket eingebettet werden, fassen wir jeweils unter dem Begriff *Timestamp* zusammen.

Es ist anzumerken, dass auf dem BTnode laufende Anwendungen die Konstante `BT_MAX_ACL_COM_PAYLOAD` unbedingt beachten sollten, da diese je nach verwendetem Algorithmus zur Compilezeit mehr oder weniger reduziert wird und die Anwendung folglich weniger Nutzdaten in einem Paket versenden darf.

Weitere Möglichkeiten und Erwägungen werden auf Seite 38 in Abschnitt 3.5 erläutert.

3.3.5 Ankerknoten

Die Uhren der BTnodes sollen in unserer Implementierung extern synchronisiert werden, d.h. sie müssen nicht einfach untereinander synchron laufen, sondern sollen die Zeit "von aussen" beziehen und so die reale Zeit angeben können. Dafür müssen gewisse BTnodes eine Verbindung zu einem Zeitgeber haben. Diese BTnodes werden als Ankerknoten bezeichnet (während die übrigen als Sensorknoten bezeichnet werden) und sind über eine serielle Schnittstelle mit einem PC verbunden, der die aktuelle Zeit via NTP bezieht. Die Kommunikation zwischen PC und Ankerknoten läuft über eine UART-Schnittstelle. Ankerknoten speichern keine Zeitschranken, sondern holen sich, wann immer nötig, die Zeitinformationen vom PC. Wird also ein Paket vom Ankerknoten verschickt, so fordert dieser beim PC einen Zeitstempel an. Dieser wird vom PC direkt im richtigen Format geliefert, so dass

[11] In unserer Version 4 Byte; vgl. Abschnitt 3.3.7.

der Ankerknoten nur noch $t^{N_i}_{localdelay}$ einfügen muss, falls eine *Online*-Messung der "round-trip time" durchgeführt wird.

Ankerknoten kennen die meisten Funktionen aus der Ebene der Synchronisationsfunktionen nicht, was aber ohne Auswirkung ist, da dies ohnehin interne Funktionen sind. Sie werden ersetzt durch *function stubs*, welche dieselben Schnittstellen haben, unter denen sich aber keine Berechnungsfunktionen, sondern ein Kommunikationsprotokoll mit dem PC verbergen.

Auf dem PC muss dafür das Programm `tstampserver` laufen, welches auf der entsprechenden seriellen Schnittstelle auf die Anforderung vom Ankerknoten wartet, die aktuelle PC-Zeit in einen Zeitstempel umwandelt und diesen zurückschickt. Die Unsicherheit, welche bei dieser Kommunikation ins Spiel kommt, wird bei der Schrankenbildung auf dem PC berücksichtigt. Die Messungen aus Abschnitt 4.2.2 haben gezeigt, dass der Delay zwischen PC und Ankerknoten in den meisten Fällen 16 ms beträgt. Die auf 1 ms gerundete Standardabweichung beträgt 1ms, weshalb die untere Schranke aus der um $16 - 1 = 15$ ms und die obere aus der um $16 + 1 = 17$ ms erhöhten aktuellen PC-Zeit gebildet wird. Damit liefert ein Ankerknoten Zeitinformationen mit einer fixen Unsicherheit von 2 ms.

Obwohl Ankerknoten gemäss unserer Entwurfsziele identische Schnittstellen wie Sensorknoten haben sollten, ist bei Ankerknoten zu beachten, dass auf ihnen entweder kein Terminal (oder andere Anwendungen, welche die `USART0` verwenden) laufen sollte oder dieses niedrigere Priorität haben muss als die Threads, welche Funktionen aus `sync.c` ausführen können. Andernfalls "schluckt" das Terminal den vom PC gesendeten Zeitstempel, bevor er die Synchronisationsfunktionen erreicht.

3.3.6 Kombination zweier Zeitschranken

Die Kombination zweier Zeitschranken wird durch die Algorithmen vorgegeben. Es werden jedoch keine Angaben darüber gemacht, wie vorgegangen werden soll, falls die erhaltenen Schranken ein zu dem aktuellen Unsicherheitsintervall disjunktes Intervall bilden. In der Theorie kann dieser Fall gar nie vorkommen, da die Referenzzeit für alle Knoten garantiert innerhalb des Intervalls liegt. In der praktischen Umsetzung könnte es jedoch sein, dass wegen eines *Fehlers* (unerlaubter Speicherzugriff, Überlauf, etc.) disjunkte Intervalle entstehen. Weil ein Schneiden mit solchen Intervallen fatal wäre (die obere Schranke wäre danach kleiner als die untere; allfällige Applikationen, die das Unsicherheitsintervall mit vorzeichenlosen Variablen berechnen wollen, würden Überläufe oder undefinierte Werte erzeugen), fangen wir diese Möglichkeit ab, obwohl sie strenggenommen nie auftreten dürfte. Wichtig ist im Sinne der geforderten Robustheit auch, dass Knoten mit falschen, dis-

junkten Zeitintervallen wieder "den Anschluss finden".

In der Implementierung von BP-ISA verwerfen wir Schranken, deren Intervalle disjunkt zu den von den gespeicherten Schranken aufgespannten Intervallen sind, um die Unsicherheit nicht unnötig zu vergrössern. Die Knoten mit falschen Schranken können dann solange nicht erfolgreich synchronisieren, bis ihre Unsicherheit so stark gewachsen ist, dass ihr Zeitintervall wieder mit demjenigen von Nachbarknoten überlappt. Das ist in endlicher Zeit der Fall, da ihre Unsicherheit in diesem Fall streng monoton wachsend und der darstellbare Zeitbereich in unserem Fall endlich ist.

Für die Implementierung von IM ist unsere Strategie leicht unterschiedlich: Wenn wir ein zu unseren Schranken disjunktes Schrankenpaar erhalten, dessen untere Schranke grösser als unsere obere ist, so bilden wir die Vereinigung der beiden von den Schranken aufgespannten Intervallen und speichern deren Schranken als unsere neuen. Im anderen Fall (wenn also die obere empfangene Schranke unter unserer unteren liegt) verwerfen wir die Zeitinformation, da erstens die untere Schranke streng monoton wachsend bleiben soll (vgl. die Funktion `rt_get_monotonic()`) und wir zweitens davon ausgehen, dass wahrscheinlich die empfangene Information und nicht unsere gespeicherte falsch ist. Das ist in der Überlegung begründet, dass ein Fehler (zum Beispiel ein Überlaufen der lokalen Uhr, die vom Typ `u_long` ist) im Allgemeinen eher zu tiefe als zu hohe Schranken zur Folge hat.

Dieser kleine Implementierungsunterschied sollte sich beim Vergleich der Algorithmen nicht auswirken, da der inkorrekte Fall disjunkter Zeitintervalle gar nie auftreten dürfte.

3.3.7 Nachrichtenverzögerung

Das Problem der Unsicherheit über die Nachrichtenverzögerung wurde in dieser Arbeit auf zwei verschiedene Arten angegangen. In der ersten Variante wurde die Verzögerung der Nachrichten auf tiefer Systemebene gemessen und daraus eine Statistik erstellt (siehe Abschnitt 4.2.1). Anhand dieser Daten sollte der Empfänger mittels linearer Regression die Nachrichtenverzögerung aus der Paketlänge berechnen und bei der Schrankenbildung entsprechend berücksichtigen. Es stellte sich heraus, dass dieser Ansatz nicht sehr gut funktionierte. Die starken Schwankungen in der Verzögerung führten dazu, dass die Referenzzeit auf den Knoten nur sehr selten zwischen den beiden Schranken lag und die grundlegende Idee intervallbasierter Algorithmen (nämlich Gültigkeit der Bedingung 2.2) damit verletzt war.

Aus diesem Grund wurde beschlossen, die Nachrichtenverzögerung durch Messung der "round-trip time" (RTT) bei der Schrankenbildung einzubeziehen. Das Verfahren wird in [4] unter "round-trip synchronization" genau

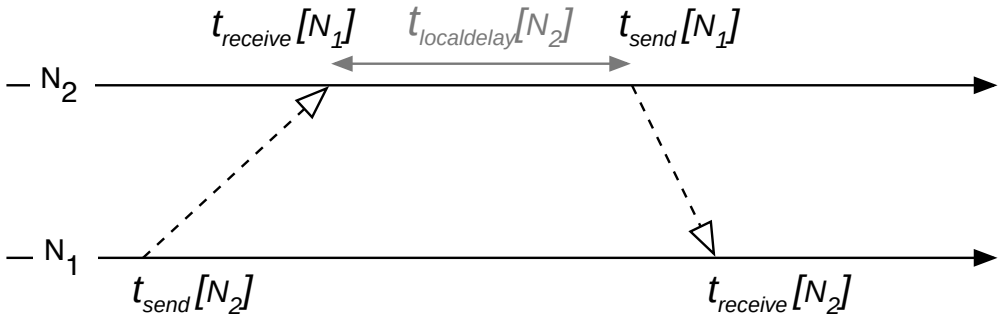

Abbildung 3.3: Messung der "round-trip time".

beschrieben. Die Funktionsweise in unserer Implementierung soll nachfolgend mit einem Beispiel erläutert werden (siehe Abbildung 3.3). BTnode N_1 sendet ein Paket an BTnode N_2 und speichert den Wert seiner lokalen Uhr in $t_{send}[N_2]$. N_2 speichert den Zeitpunkt des Empfangs mithilfe seiner lokalen Uhr in $t_{receive}[N_1]$. Später sendet auch N_2 ein Paket an N_1, wobei N_2 diesen Zeitpunkt als $t_{send}[N_1]$ ebenso speichert wie N_1 den Empfangszeitpunkt des Paketes als $t_{receive}[N_2]$. Die Zeit $t_{localdelay}[N_2] = t_{send}[N_1] - t_{receive}[N_1]$ wird der Nachricht beigefügt. N_1 kann nun die "round-trip time" berechnen: $RTT = t_{receive}[N_2] - t_{send}[N_2] - t^{N_2}_{localdelay}$. Diese RTT kann entweder verwendet werden, um die in einem Paket enthaltene Zeitinformation zu verwerfen, falls RTT grösser als ein durch ein Parameter des Algorithmus gegebener Wert ist, oder aber die RTT fliesst in die Berechnung des Alters der Information ein. Dabei wird angenommen, dass eine minimale Nachrichtenverzögerung $delay_{min}$ existiert, welche nicht unterschritten wird. Den entsprechenden Wert haben wir durch Messungen ermittelt.[12] Die von N_2 erhaltene Information ist also im Minimum $delay_{min}$ alt, weshalb die untere Schranke dieser Nachricht um gerade diesen Wert erhöht wird. Die obere Schranke für das Alter der Information erhält man, indem man annimmt, dass die Nachricht von N_1 zu N_2 (also "auf dem Hinweg") nur um $delay_{min}$ verzögert worden sei. Daraus ergibt sich, dass die Antwort von N_2 an N_1 höchstens um $RTT - delay_{min}$ verzögert worden sein kann. Zur oberen Schranke der enthaltenen Zeitinformation wird deshalb $RTT - delay_{min}$ addiert.

Zu beachten ist, dass für die erwähnten Messungen die Uhren auf den

[12]Eine *sichere* untere Schranke (z.B. durch Auszählen der nötigen Prozessorzyklen) konnten wir in Unkenntnis der internen Vorgänge im Bluetooth-Modul und wegen der nicht-deterministischen Eigenschaften der Nachrichtenübertragung (Threadswitches, Retransmissions) nicht ermitteln.

beiden BTnodes (noch) nicht synchronisiert sein müssen, da nur Differenzen aus Zeitwerten, welche auf einem einzigen BTnode gemessen werden, gebildet werden.[13] Auf den BTnodes bietet es sich deshalb an, die entsprechenden Zeiten durch die Anzahl vergangener Sekunden seit Programmstart zu repräsentieren. Da diese Zeiten für jeden einzelnen BTnode, mit welchem kommuniziert wird, festgehalten werden müssen, werden sie in einer Liste zugeordnet zu der Bluetooth-Adresse des Kommunikationspartners gespeichert.

Nachfolgend werden die beiden implementierten Varianten mit "Verzögerungs-Statistik" und "RTT-Messung" auseinandergehalten.

3.4 Kommentar zum Quellcode

Die Synchronisationsfunktionen wurden in den Dateien

- `btnut/btnode/include/sync/globals.h`
- `btnut/btnode/include/sync/sync.h`
- `btnut/btnode/include/sync/sync-anchor.h`
- `btnut/btnode/sync/sync.c`
- `btnut/btnode/sync/sync-anchor.c`

implementiert. Wo nötig, wurden bereits vorhandene Systemdateien angepasst.

3.4.1 Präprozessor-Konstanten

Wahl der Implementierungsvariante: Über Präprozessor-Konstanten kann die Synchronisation leicht ein- und ausgeschaltet, der verwendete Algorithmus und die Art des Knotens (Sensorknoten oder Ankerknoten) festgelegt werden. Die entsprechenden `#define`-Zeilen sind in der Datei `globals.h` zu finden.

- Mit `#define SYNC` werden die Synchronisationsfunktionen "eingeschaltet".

[13] Genau genommen müssten auch diese lokalen Zeitwerte als Schrankenpaar dargestellt werden, da die Uhren der kommunizierenden Knoten unterschiedliche Drift haben. Unter der Annahme einer Kommunikationshäufigkeit von mindestens $\frac{6}{h}$ im schlechtesten Fall (maximal unterschiedliche Drift) bleibt der Fehler aber in der Grössenordnung der normalen Schwankungen der RTT-Werte (ca. 60ms).

- Mit `#define BP_ISA` wird BP-ISA als Algorithmus verwendet. Wird diese Zeile auskommentiert, wird IM verwendet.

- Mit `#define IS_ANCHORNODE` wird festgelegt, dass es sich um einen Ankerknoten handelt. Für einen Sensorknoten ist diese Zeile auszukommentieren.

Einstellbare Parameter: Ebenfalls mit Präprozessor-Konstanten lassen sich diverse Parameter einstellen. Sie finden sich in der Datei `sync.h` und sind dort auch näher beschrieben. Unter anderem finden sich dort folgende Parameter:

- `DRIFT_CONSTRAINT` gibt die maximale Drift in ppm (parts per million) an und entspricht ρ_{max} im Kapitel 2. Der Wert kann zum Beispiel angepasst werden, falls man A-priori-Wissen über die klimatischen Verhältnisse hat, in denen der BTnode zum Einsatz kommt. Bei unseren Tests haben wir eine maximale Drift von ± 65 ppm eingesetzt. Im Datenblatt [8] sind zwar ± 20 ppm maximale Drift bei 25 °C, ± 3 ppm/Jahr Alterung, ± 8 ppm "shock resistance" sowie eine Formel für die Temperaturabhängigkeit angegeben. Um korrekte Schranken garantieren zu können, haben wir einen etwas grösseren Wert eingesetzt.

- Mit `DRIFT_ADJUST` kann die Verteilung der maximalen Drift ungleichmässig eingestellt werden, falls man A-priori-Wissen über die Drift der Uhr gegenüber der Referenzzeit hat. Der Parameter nimmt Werte zwischen 0 und 1 an und gibt den Faktor an, mit dem der gesamte Unsicherheitszuwachs der oberen Schranke zugeschlagen werden soll. Im Normalfall beträgt `DRIFT_ADJUST` also 0.5. In unserer Testanordnung verwendeten wir den Wert 0.75, da wir beobachtet hatten, dass die Uhren der BTnodes in allen vorbereitenden Messungen zu langsam liefen. Auch diente der Parameter dazu, grössere Driftunterschiede zwischen den Netzknoten zu simulieren.

- `SYNC_MAX_RTT` gibt die obere Schranke für die "round-trip time" (RTT). Die Zeitinformation aus Paketen mit einer längeren gemessenen RTT wird nicht verwendet, da dann die Unsicherheit über die Nachrichtenverzögerung, die sich durch Subtraktion der minimalen von der gemessenen RTT direkt ergibt[14], zu gross ist.

[14] Zur Berechnung des Unsicherheitswertes vgl. Abschnitt 3.3.7, zur Bestimmung der minimalen RTT vgl. Abschnitt 4.2.1.

- Mit `SYNC_MAX_NEIGHBOURS` wird angegeben, wie viele Nachbarknoten maximal in der Liste geführt werden sollen, die für BP-ISA und IM mit "RTT-Messung" nötig ist. Hat die Länge der Liste diesen Wert erreicht, so wird das letzte Element (also der älteste Eintrag, da Elemente stets zuvorderst eingefügt werden) gelöscht, falls ein neues eingefügt werden soll. Obwohl ein Knoten in einem Bluetooth-Piconetz nur 7 aktive Nachbarn haben kann, ist es sinnvoll, den Wert grösser zu wählen, wenn man von mobilen oder nur zeitweise aktiven[15] Knoten ausgeht, was ja in Ad-hoc-Netzen der Fall ist.

Nach einer Änderung dieser Konstanten sollte das ganze BTnut nochmals neu kompiliert werden. Dadurch werden auch die von uns modifizierten Systemdateien mit den neuen `#define`-Anweisungen korrekt kompiliert.

In der nachfolgenden Beschreibung ist zu beachten, dass es sich dabei um die Implementierungs-Variante "RTT-Messung" handelt. Für Details sei auf den Quellcode verwiesen. Der Code ist zwar für alle Varianten ausführlich kommentiert, die zahlreichen Präprozessoranweisungen für alle möglichen Kombinationen von Implementierungsvarianten erschweren die Leserlichkeit jedoch stark. Es empfiehlt sich, den Compiler diese Anweisungen in der gewünschten Form auswerten zu lassen und dann den so erhaltenen Code zu betrachten.[16]

3.4.2 Datentypen

Folgende Datentypen wurden neu eingeführt:

- Die Struktur `syncbounds` speichert eine obere und untere Schranke, deren Berechnungszeitpunkt in lokaler Zeit (`last_update`) sowie die Gültigkeit dieser Schranken:

```
typedef struct {
    u_longlong lower;
    u_longlong upper;
    u_long last_update;
    char validity;
} syncbounds;
```

- `timeval` ist die Implementierung des gleichnamigen C-Datentyps und repräsentiert die Unix-Zeit in Sekunden und Mikrosekunden. Sie wird als Rückgabewert der Funktion `gettimeofday()` verwendet.

[15] Ein Piconetz kann bis 256 Knoten beherbergen, von denen jedoch höchstens 8 (1 Master und 7 Slaves) gleichzeitig aktiv sein können.

[16] Die entsprechenden `gcc`-Flags wären z.B.: `-E -C -P -imacros <h-files>`

- Objekte vom Typ `sync_neighbour` speichern jeweils die Schranken und lokalen Zeitpunkte der letzten Kommunikation mit dem entsprechenden BTnode:

  ```
  typedef struct _sync_nb sync_neighbour;

  struct _sync_nb {
      bt_addr_t address;
      syncbounds bounds;
      sync_neighbour * next;
      u_long last_rcvd;
      u_long last_send;
      u_char uncertainty;
  };
  ```

  Aus diesen Elementen wird für BP-ISA und IM mit "RTT-Messung" eine verkettete Liste aufgebaut. Die BTnodes werden anhand ihrer Bluetooth-Adresse identifiziert.[17] Die Liste wird immer so sortiert, dass das erste Element vom neuesten Kommunikationsereignis stammt. Die Anzahl der Elemente in der Liste ist – wie oben beschrieben – durch die Konstante `SYNC_MAX_NEIGHBOURS` beschränkt.

3.4.3 Globale Variablen

Die Struktur `sync_globals` umfasst alle für die Uhrensynchronisation nötigen globalen Variablen:

```
struct _sync_globals_struct {
    u_char sync_timestamp_to_send[SYNC_TSTAMP_LEN + 1];
    u_char sync_timestamp_rcvd[SYNC_TSTAMP_LEN + 1];
    sync_neighbour * nearest_neighbour;
    u_char neighbour_count;
    bt_addr_t neighbour_to_send_addr;
    char bitpattern;
} sync_globals;
```

Das Zusammenfassen der globalen Variablen erlaubt schnelleren Zugriff, da bloss die Basisadresse in einem Register liegt und Speicherzugriffe mit indi-

[17] Auch andere Identifikationsmittel wären denkbar: 1.) `module_con_handle`; 2.) `app_con_handle`; 3.) eine eigene Erkennung. Die Bluetooth-Device-Adresse vereint aber ideal Eindeutigkeit ("application uniqueness") und Eintrittsinvarianz ("reentrancy").

rekter Adressierung erfolgen.[18] Die Variablen werden folgendermassen verwendet:

- `u_char sync_timestamp_to_send[SYNC_TSTAMP_LEN+1]` dient dazu, einen aktuellen Zeitstempel zwischenzuspeichern, bevor er in der Funktion `bt_hci_transport_uart()` übertragen wird.
- In `u_char sync_timestamp_received[SYNC_TSTAMP_LEN+1]` wird ein empfangener Zeitstempel abgespeichert, damit er von den Synchronisationsfunktionen verarbeitet werden kann.
- `sync_neighbour * nearest_neighbour` zeigt auf das erste Element der verketteten Liste der benachbarten BTnodes. Die aktuellen Schranken werden bei Bedarf aus den in diesem Element gespeicherten Schranken berechnet.
- `u_char neighbour_count` speichert die aktuelle Länge der Liste.
- `bt_addr_t neighbour_to_send_addr` speichert die Adresse, an welche ein Paket gesendet wird.
- `char bitpattern` wird verwendet um zu definieren, welche LEDs beim "Heartbeat"[19] aufblinken sollen. Dieses Muster ist zum Beispiel bei Ankerknoten unterschiedlich, um sie als solche schnell erkennen zu können.

Weiter wird die globale Variable `u_long btn_led_synch_offset` verwendet, um den "Heartbeat" auf mehreren Sensorknoten gleichzeitig aufblinken zu lassen. Zu beachten ist, dass ein Ankerknoten *nicht* synchron zu den übrigen Sensorknoten blinken wird, da er Zeitinformationen einfach weiterleitet und deshalb nie im Besitz von Schranken ist.

3.4.4 Senden eines Paketes - Bluetooth-Funktionen

Nachfolgend werden die Funktionsaufrufe beschrieben, welche beim Versenden eines Datenpakets erfolgen. Der Ablauf ist in Abbildung 3.4 dargestellt. Eine Applikation ruft in diesem Fall die Funktion `bt_acl_com_send_packet()` aus `bt_acl_com.c` auf. Sie übergibt unter anderem das "connection handle", welche die erstellte Verbindung eindeutig identifiziert, einen Zeiger auf die zu versendenden Daten, sowie die Länge der Daten. Diese Länge

[18] Dies ist sehr effizient, da der Mikrokontroller in der Ladeinstruktion `LD` den Offset zur Basisadresse im selben Zyklus addieren bzw. subtrahieren kann (vgl. [6]).

[19] Ein periodisch aufblinkendes LED-Muster, welches anzeigt, dass der BTnode noch "lebt".

wird als erstes um `SYNC_TSTAMP_LEN` erhöht, da den Daten der Zeitstempel vorangestellt wird. Im Falle von BP-ISA oder IM mit "RTT-Messung" wird nun aus dem "connection handle" die Bluetooth-Adresse bestimmt und in die globale Variable `neighbour_to_send_addr` gespeichert. Mit dieser Information kann später bestimmt werden, welche gespeicherten vergangenen Schranken zusätzlich zu den aktuellen mitgesendet werden müssen bzw. in welchem Listenelement Informationen über den Sendezeitpunkt gespeichert werden sollen. Danach werden wie üblich die Felder `service_nr` und `len` des BT-ACL-COM-Headers gesetzt und die Funktion `bt_hci_send_acl_pkt()` aus `bt_hci.c` wird aufgerufen, welche unter anderem den Header des `HCI_ACL_DATA_PACKET`-Paketes setzt. Das erste Byte dieses Headers dient zur Unterscheidung der verschiedenen HCI-Pakettypen und wird entsprechend auf `HCI_ACL_DATA_PACKET` gesetzt. Daneben sind unter anderem Felder für das "connection handle" und die Gesamtlänge der Daten vorhanden. Das zu versendende Paket bzw. ein Zeiger darauf wird im Anschluss an die Funktion `_bt_hci_send_pkt()` aus `bt_hci_transport_uart.c` weitergereicht, welche zunächst die verschiedenen Pakettypen unterscheidet – diese Funktion wird im Gegensatz zu den vorhin erwähnten auch zum Versenden der anderen Pakettypen verwendet – und die Länge der Daten aus dem Header liest. Daraus wird durch Hinzuaddieren der Header-Länge die totale Anzahl der auf die UART-Schnittstelle zu schreibenden Bytes berechnet. Für die Uhrensynchronisation wird nun überprüft, ob es sich um ein `HCI_ACL_DATA_PACKET`-Paket handelt. Falls dies zutrifft, muss ein aktueller Zeitstempel zum Versenden generiert werden, wozu die Funktion `sync_generate_timestamp()` aus `sync.c` aufgerufen wird. Im Falle von BP-ISA oder IM mit "RTT-Messung" wird das zusätzlich benötigte Argument `neighbour_to_send_addr` übergeben. Danach kann das gesamte Paket über die UART-Schnittstelle zum Bluetooth-Modul geschickt werden. Handelt es sich um ein Datenpaket, so wird nach dem Schreiben des ACL-Headers und des BT-ACL-COM-Headers der Zeitstempel, welcher in der globalen Variable `sync_timestamp_to_send` abgelegt ist, geschrieben, danach der restliche Teil des Paketes. Die anderen Pakettypen werden direkt – so wie sie übergeben wurden und ohne Zeitstempel – übertragen.

3.4.5 Senden eines Pakets - Synchronisation

Wie erwähnt wird kurz vor dem Schreiben auf die UART-Schnittstelle zum Bluetooth-Modul die Funktion `sync_generate_timestamp()` aus `sync.c` aufgerufen. Sie bewirkt, dass ein aktueller Zeitstempel in die globale Variable `sync_timestamp_to_send` geschrieben wird. Im Fall BP-ISA wird zusätzlich die Liste der Nachbarn aktualisiert: Falls der Empfänger des Pakets noch

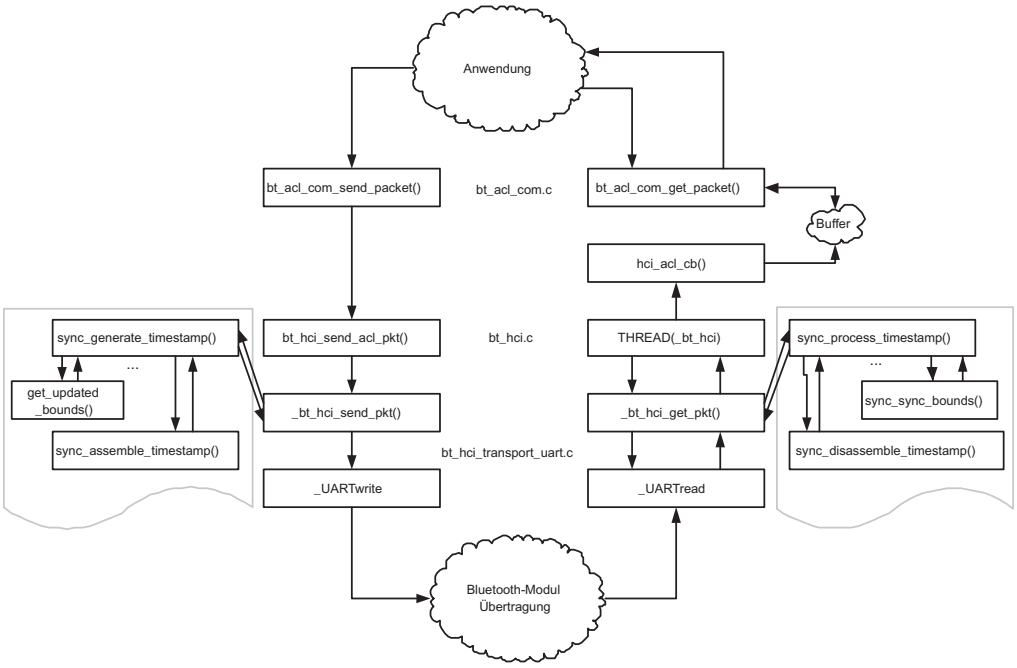

Abbildung 3.4: Ablauf beim Empfangen und Versenden von Paketen (vereinfachte Darstellung).

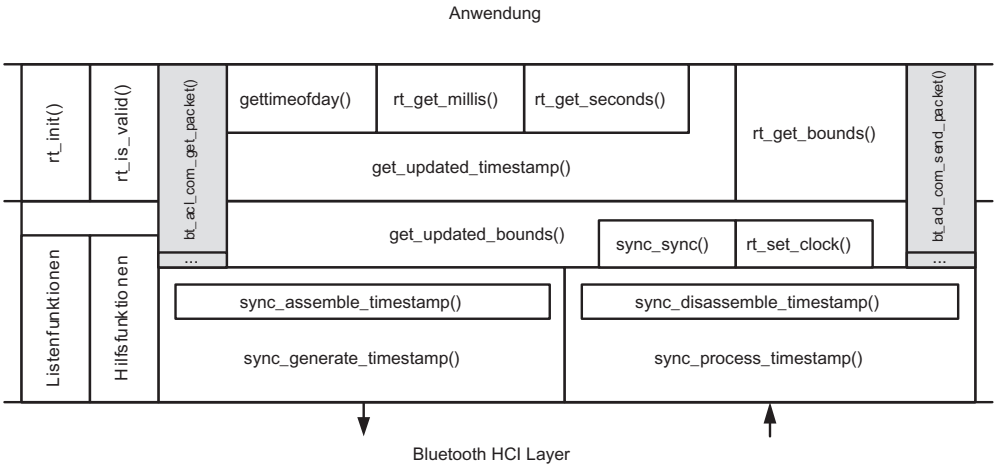

Abbildung 3.5: Übersicht über implementierte Funktionen.

nicht Teil der Liste ist, wird ein entsprechendes Element hinzugefügt, und schliesslich wird die in 3.2.3 vorgeschlagene zusätzliche Aktualisierung des Listeneintrags des Empfängers vollzogen.

Auf einem Ankerknoten wird der Aufruf direkt an die Funktion `sync_anchor_get_timestamp()` aus `sync-anchor.c` weitergereicht, welche einen Zeitstempel vom PC anfordert. Auf Sensorknoten werden die aktuellen Schranken über `get_updated_bounds()` berechnet und danach mit `sync_assemble_timestamp()` in das definierte Zeitstempel-Format umgewandelt. Die Hilfsfunktion `lltobs()` wandelt dabei die als `u_longlong` gespeicherten Schranken in einen Bytestream um. Bei BP-ISA oder IM mit "RTT-Messung" wird zusätzlich nach dem Eintrag des Empfängerknotens in der verketteten Liste gesucht. Die für die "RTT-Messung" benötigte Zeit zwischen Empfang des letzten Pakets vom Empfängerknoten und der aktuellen Zeit kann berechnet und in den Zeitstempel integriert werden. Für BP-ISA werden zusätzlich die Schranken aus der letzten Kommunikation beigefügt. Nun enthält `sync_timestamp_to_send` den aktuellen Zeitstempel.

3.4.6 Schrankenberechnung

`get_updated_bounds()` ist im Wesentlichen die Umsetzung der Formel 2.3 aus Kapitel 2.2 und berechnet die aktuellen Schranken aus den im ersten Listenelement gespeicherten. Da insbesondere im Fall von BP-ISA diese Funktion oft aufgerufen wird, wurde die Implementierung besonders sorgfältig vorgenommen. So wurde für Δh bewusst eine vorzeichenbehaftete 64-Bit-Variable gewählt, da die Differenz zwischen `reference_time` $\in (0, 2^{32})$ und `last_update` $\in (0, 2^{32})$ im Intervall $(-2^{32}, 2^{32})$ liegen kann. Die Divisionen der Formel 2.3 wurden vermieden, indem entsprechende Faktoren zur Compilezeit vorberechnet werden, sodass nur noch ein 20-Bit-Rightshift nötig ist. Um korrekt zu runden, wird zuvor das grösste (most significant), später vernachlässigte Bit hinzuaddiert. Dies ist äquivalent zu einem Runden nach dem Shift. Ausserdem wurden verschiedene Varianten geschrieben[20] und diese bezüglich Codegrösse, Framegrösse und geschätztem Zyklenbedarf sowie anhand der Kriterien feste Laufzeit, Portabilität und arithmetische Genauigkeit evaluiert.

3.4.7 Empfang eines Pakets - Bluetooth-Funktionen

Empfangsseitig wird beim Initialisieren des Bluetooth-Stacks der Thread `bt_hci_thread` aus `bt_hci.c` gestartet, welcher in einer Schleife jeweils

[20] Runden mittels Addition oder Inline-Assembler-Bit-Test, Berechnung in eigener Funktion oder integriert, Argumentübergabe *per value* oder mit Zeiger etc.

`_bt_hci_get_pkt()` aus `bt_hci_transport_uart.c` aufruft. Diese Funktion liest von der UART-Schnittstelle zunächst das erste Byte, welches den Pakettyp identifiziert. Im Falle eines ACL-Datenpaketes werden danach die ACL- und BT-ACL-COM-Header gelesen und die Länge der Payload extrahiert. Danach kann der eingebettete Zeitstempel gelesen und in `sync_timestamp_rcvd` gespeichert werden. Die restlichen Daten der Payload werden dann vom Bluetooth-Modul gelesen. Bei BP-ISA oder IM mit "RTT-Messung" wird nun noch aus dem "connection handle" die Bluetooth-Adresse des Senders ermittelt. Mit dem Aufruf von `sync_process_timestamp()` aus `sync.c` werden die Synchronisationsfunktionen in der Folge in Gang gesetzt. Darauf wird die extrahierte Datenlänge um `SYNC_TSTAMP_LEN` reduziert, da der Timestamp nun wieder entfernt wurde bzw. nicht in das zurückgegebene Paket geschrieben wurde. Der Thread ruft danach den Callback `hci_acl_cb` auf, welcher das Paket in einen Puffer einfügt und ein Event auslöst, um zu signalisieren, dass ein Paket angekommen ist. Mit `bt_acl_com_get_packet()` erhält eine Anwendung einen Zeiger auf das gespeicherte Paket zurück.

3.4.8 Empfang eines Pakets - Synchronisation

Die Funktion `sync_process_timestamp()` extrahiert zunächst mithilfe von `sync_disassemble_timestamp()` die 2 (IM) bzw. 4 (BP-ISA) Schranken aus dem Zeitstempel und ruft dann `sync_sync_bounds()` auf. Sind die empfangenen Schranken ungültig, so werden sie ignoriert. Sind die bisher gespeicherten Schranken noch ungültig, so wird `rt_set_clock()` aufgerufen, um die erhaltenen Schranken zu speichern. Sind die empfangenen sowie die bereits gespeicherten Schranken gültig, so müssen diese wie im Algorithmus beschrieben kombiniert werden. Bei BP-ISA müssen einige zusätzliche Schritte ausgeführt werden: Falls der sendende BTnode noch keinen Eintrag in der verketteten Liste hat, muss ein neuer erstellt werden. Ist bereits ein Eintrag vorhanden, so wird das entsprechende Element aus der Liste entfernt, um während des Ausführens der Synchronisationsfunktionen gesondert behandelt und später wieder zuvorderst in die Liste eingefügt zu werden.

3.4.9 Synchronisationsfunktionen

Die tatsächliche Synchronisation, also die Berechnung der besten aktuellen Schranken aus empfangener und gespeicherter Information gemäss den in Abschnitt 2.2 beschriebenen Algorithmen, findet in der Funktion `sync_sync_bounds()` statt. Der Ablauf entspricht in beiden Fällen (IM und BP-ISA) weitgehend der Beschreibung in Abschnitt 2.2.1 beziehungsweise 2.2.2. Im Fall BP-ISA ist zu beachten, dass die `bounds.last_update` gespeicherten

Zeitpunkte auf beiden kommunizierenden Knoten nicht denselben tatsächlichen Zeitpunkt angeben, da echte Gleichzeitigkeit ja wie in Abschnitt 3.2.3 dargelegt nicht möglich ist. Wir verwenden also auch hier den aus "round-trip time" ermittelten Unsicherheitswert: Für das Kombinieren der aktuellen Schranken `current_bounds_rcvd` den vom gerade geschehenen, auslösenden Kommunikationsereignis verursachten Unsicherheitswert, beim Kombinieren der Schranken `previous_bounds_rcvd` den Unsicherheitswert der damaligen Kommunikation, der in der Eigenschaft `uncertainty` des entsprechenden Listenelements gespeichert ist. Schliesslich wird der momentane Unsicherheitswert in dieser Variablen gespeichert, damit er bei der nächsten Kommunikation mit dem kommunizierenden Nachbarknoten wieder verwendet werden kann.

3.4.10 Highlevel-Funktionen

Folgende Funktionen stehen den Anwendungen neu zur Verfügung:

- `rt_init()` muss von einer Anwendung zu Beginn aufgerufen werden, um die Synchronisationsfunktionalität zu initialisieren.

- `gettimeofday()` ist die BTnode-Implementierung der gleichnamigen POSIX-Standardbibliotheksfunktion[21], welche die Zeit im Unix-Format angibt. Die Zeit wird dabei aus dem Mittel der unteren und oberen Schranke gebildet. Bei Aufruf werden die Anzahl Sekunden seit dem 1. Januar 1970, sowie die Anzahl Mikrosekunden seit der letzten Sekunde in einem `struct` vom Datentyp `timeval` gespeichert. Da die Granularität der Synchronisationsfunktionen jedoch im Millisekunden-Bereich liegt, werden die letzten drei Ziffern der Mikrosekunden immer Null sein. Aus Kompatibilitätsgründen wurde das Format des Datentyps `struct timeval` beibehalten. Der Rückgabewert der Funktion ist 0, wenn der BTnode über gültige Schranken verfügt, andernfalls -1. Falls die Schranken nicht gültig sind, hat der BTnode noch keine Zeitinformationen von anderen BTnodes erhalten.

- `syncbounds rt_get_bounds( void )` liefert die aktuellen Schranken zurück.

- `rt_get_monotonic( u_longlong * millis )` liefert eine streng monoton wachsende Zeit, die garantiert im Echtzeitintervall liegt. Im Moment gibt sie einfach die untere Schranke zurück. Dadurch sind zwar die

[21] Normalerweise in `sys/time.h` deklariert; die Funktion ist aber *keine* POSIX-konforme Implementierung, da ja auch BTnut diese Spezifikation nicht erfüllt.

beiden Kriterien erfüllt,[22] der Fehler ist im Durchschnitt aber maximal (vgl. für andere Möglichkeiten Kapitel 3.5).

- `void rt_get_average( long long * av )` schreibt den Mittelwert der aktuellen Schranken in Millisekunden in die mit `av` referenzierte Variable.
- Mit `int rt_is_valid( void )` kann die Gültigkeit der Schranken überprüft werden. Die Funktion gibt 1 zurück, falls der BTnode im Besitz von gültigen Schranken ist, ansonsten 0.
- `u_long rt_get_millis( void )`, `u_long rt_get_seconds( void )` haben als Rückgabewert das Mittel der aktuellen Schranken in Millisekunden bzw. Sekunden.

3.4.11 Kommunikation Ankerknoten – PC

Zwischen Ankerknoten und PC läuft ein simples Kommunikationsprotokoll über die UART-Schnittstelle ab. Der `tstampserver` wartet auf ein `REQ_CHAR`-Zeichen vom Ankerknoten, generiert einen Zeitstempel und schickt ihn dem Ankerknoten. Dieser bestätigt den Empfang mit einem `ACK_CHAR` Zeichen. Wichtig hierbei ist, dass kein anderes Programm die UART-Schnittstelle belegt und dass die Übertragungsgeschwindigkeit für `tstampserver` und Ankerknoten gleich gesetzt ist.

3.4.12 PC-Programme

Für den PC wurden die Programme `tstampserver` und `btncontrol` geschrieben. Beide können mit `program_name [device number [baud rate]]` gestartet werden. `device number` bezeichnet hierbei die Nummer des Schnittstellen-Devices und wird an `/dev/ttyUSB` angehängt. Es können Werte von 0 bis 9 angegeben werden, wobei 9 `/dev/stdin` als Device wählt, um Debugging mittels manueller Eingabe zu erlauben. Mit `baud rate` kann die Verbindungsgeschwindigkeit angegeben werden. Erlaubte Werte sind 115 für Kommunikation mit 115200 bit/s und 576 für 57600 bit/s.[23] Standardmässig wird die Verbindung zum BTnode über `/dev/ttyUSB0` mit 115200 bit/s erstellt.

`tstampserver` sendet dem Ankerknoten auf Anfrage einen aktuellen Zeitstempel.

[22] Die Monotonität leitet sich aus der Schnittregel 2.4 ab.

[23] Wir haben für beide Raten dreistellige Kürzel definiert, da dies das zusammen mit den BTnodes oft verwendete Terminal-Programm `minicom` ebenso tut.

`btncontrol` kann dazu genutzt werden, die Schranken in gleichmässigen Intervallen vom BTnode auszulesen oder lokal abspeichern zu lassen. Hierzu muss das Programm `sync-test` auf dem BTnode laufen.

3.4.13 BTnode-Testapplikation `sync-test`

Für verschiedene Funktionstests und Messungen wurde für den BTnode das Programm `sync-test` geschrieben. Es erlaubt einerseits das Ausgeben der Zeitinformationen direkt in einem Terminal, stellt aber auch Funktionen zur Verfügung, auf welche der PC zurückgreifen kann, um Daten (Schranken, Nachbarlistenzustand etc.) auszulesen oder auf dem BTnode zu bestimmten Zeitpunkten abspeichern zu lassen. Dadurch konnten längere Messreihen ohne grossen Aufwand durchgeführt werden. Zudem kann in `sync-test` das Kommunikationsverhalten eines BTnodes definiert werden. Mit der Präprozessor-Konstante `AUTO_ANSWER` kann eingestellt werden, ob ein BTnode beim Empfang eines Pakets eine Antwort an den Sender schicken soll. Zu beachten ist, dass eine positive Rückkopplung entsteht, falls auf beiden miteinander kommunzierenden BTnodes `AUTO_ANSWER` definiert ist.

Wird `AUTO_ANSWER2` gesetzt, so wird nur auf ein Paket geantwortet, falls das erste Byte der Payload den Wert 0 hat. Im Antwortpaket wird dieses erste Byte auf 1 gesetzt. Somit kann `AUTO_ANSWER2` auf mehreren miteinander kommunizierenden BTnodes benutzt werden, um eine Zweiwegkommunikation zu simulieren ohne dass Rückkopplung entstehen kann.

Mit `AUTO_COMMUNICATE` kann ein BTnode veranlasst werden, selbständig zu kommunizieren. Dabei ist definierbar, an welche Bluetooth-Adressen Pakete verschickt werden dürfen. Es sind dabei mehrere Gruppen von Bluetooth-Adressen definierbar und über die Konstante `ALLOWED` freischaltbar. Zu beachten ist, dass `NUM_OK_ADDR` dabei immer auf die Anzahl erlaubter Adressen angepasst werden muss. Der `AUTO_COMMUNICATE`-Mechanismus wurde mit einem Thread realisiert, welcher in definierbaren Intervallen eine Bluetooth-Inquiry durchführt, um benachbarte BTnodes zu finden. Sind keine erlaubten Geräte in Reichweite, so wird die Inquiry wiederholt, bis das der Fall ist. Aus den erlaubten Geräten wird im Anschluss zufällig eines ausgewählt und ein Paket an dieses verschickt. Über die LEDs wird der Zustand des Threads angezeigt. Eine Inquiry wird durch kurzes Aufblinken der grünen LED angezeigt. Die Anzahl gefundener Geräte, mit welchen kommuniziert werden darf, blinkt danach binär codiert kurz auf. Das erfolgreiche Verbinden mit dem ausgewählten Gerät und das Versenden des Pakets wird mit einem aufsteigenden, ein Fehler mit einem absteigenden LED-Pattern angezeigt. Die Anzahl der verschickten Pakete blinkt nach dem ersten Senden binär codiert periodisch auf. Die Zeit, welche zwischen zwei Kommunikationsversuchen ver-

streichen soll, wird mit `AUTO_COM_INIT_FREQ` definiert. Zusätzlich kann mit `AUTO_RANDOM` eine maximale zufällige Abweichung von diesem Intervall eingestellt werden. Weiter ist auch definierbar, wieviele Bytes an Daten jedesmal (bei fixer Datenlänge) bzw. maximal (bei zufälliger Datenlänge) versendet werden sollen.

Weitere Funktionen erlauben das Setzen der Zeit über das Terminal, das Ausgeben der momentanen Nachbarliste, das Zurücksetzen der Echtzeituhr und Ähnliches.

3.4.14 Änderungen in anderen Dateien

Kleine Anpassungen an bestehenden Funktionen waren nötig, wobei alle Änderungen nur im kompilierten Code enthalten sind, wenn `SYNC` gesetzt ist.

led/btn-led.c Der "Heartbeat" wird, sobald ein Knoten gültige Schranken besitzt, nicht mehr bloss von der Globalen `btn_led_offset` gesteuert, sondern von der um `btn_led_synch_offset` versetzten lokalen Zeit. Ist der Wert dieser Summe ein Vielfaches von 2000, so werden die entsprechenden LED's eingeschaltet. Damit soll ein synchrones Blinken mit 0.5 Hz erreicht werden.

bt/bt_acl_com.c In der Funktion `bt_acl_com_send_packet` wird die Bluetooth-Adresse des Empfängers in der globalen Variable `sync_globals.neighbour_to_send_addr` eingetragen, da sie auf dieser Ebene noch direkt greifbar ist (später benötigt man dazu Funktionen wie `_bt_hci_get_app_con_handle()`).

bt/bt_hci_transport_uart.c Aus den Funktionen `_bt_hci_get_pkt` und `_bt_hci_send_pkt` geschehen die eigentlichen Aufrufe der Synchronisationsfunktionen, wie in Abbildung 3.4 ersichtlich ist.

nut/os/arch/avr_realtime.c Dies ist eine zusätzliche Datei, welche die Funktion `NutGetRealMillis` bereitstellt, die auch dann lokale Zeit in Millisekundengranularität gewähren soll, wenn die Variable `NUT_CPU_FREQ` nicht gesetzt ist und die Softwareclock nur mit einer Granularität von 62.5 ms inkrementiert wird. Die Funktion kombiniert die Information der Softwareclock zu diesem Zweck mit dem momentanen Wert des Hardwarecounters (`TCNT0`, vgl. [6]).

```
u_long NutGetRealMillis(void){
    u_long rc;

    NutEnterCritical();
    rc = millis + ((u_int) (TCNT0 * 125 + 512) >> 9);
    NutExitCritical();

    return rc;
}
```

Die Datei wird aus `nut/os/timer.c` eingebunden; folgerichtig ist also auch diese sowie die zugehörige h-Datei `nut/include/sys/timer.h` verändert worden.

3.5 Weitere Implementierungsmöglichkeiten

Die oben beschriebene Implementierung realisiert die Uhrensynchronisation auf relativ simple Weise. Es soll nun noch kurz auf mögliche Erweiterungen eingegangen werden, welche aus Zeitgründen leider nicht umgesetzt werden konnten. Auch alternative Ansätze, die wir bei unseren Entwurfsentscheiden diskutiert haben, werden kurz erwähnt. Folgende Punkte könnten unserer Meinung nach noch verbessert werden:

3.5.1 Gemischte Netze

Im Moment lässt sich zwar die Synchronisationsfunktionalität über die Präprozessorkonstante `SYNC` ein- und ausschalten, innerhalb eines Netzes müssen aber alle kommunizierenden BTnodes mit derselben Einstellung kompiliert sein, da sonst Payload als Zeitinformation oder umgekehrt interpretiert wird. Dies liesse sich umgehen, indem eine eigene *class of device (COD)* definiert würde. Damit könnten BTnodes beim Verbindungsaufbau erkennen, ob der entsprechende Nachbarknoten Synchronisation betreiben möchte oder nicht und sich entsprechend verhalten. Im Rahmen dieser Arbeit wurde dies noch nicht getan, unter anderem weil die Behandlung der *COD* in der Implementierung des Bluetooth-Stacks zum Zeitpunkt, als wir unsere Arbeit begannen, noch fehlerhaft war.

3.5.2 Überlauf der Lokalzeit

Obwohl das Zeitformat der Schranken innerhalb der “UNIX-Epoch” nicht überläuft, verursacht der Überlauf der Softwareclock des Nut/OS nach 49,7

Tagen Probleme, da die Werte h_M beziehungsweise `last_update` immer in lokaler Zeit angegeben sind, so dass dann insbesondere Δh falsch berechnet wird. Es lässt sich jedoch leicht Abhilfe schaffen, indem beim Initialisieren der Echtzeituhr gleich ein Timer auf den Zeitpunkt des Überlaufs gestellt wird.[24] Der entsprechende Callback würde dann alle gespeicherten Schranken ungültig setzen. (Auch `last_send` und `last_rcvd` müssten auf Null gesetzt werden.) Sobald der Knoten wieder mit Nachbarn mit gültigen Schranken kommuniziert und die RTT genügend klein ist, kommt der Knoten in Besitz neuer, gültiger Zeitinformation. Unter der Annahme, dass nicht alle Knoten in einem Netz zum selben Zeitpunkt eingeschaltet werden und deshalb auch nicht gleichzeitig überlaufen werden, sollte der Effekt keinen grossen Einfluss haben.[25]

3.5.3 Zeitformat

Für jede Schranke einen Zeitstempel mit voller Auflösung zu versenden ist zwar eine konzeptuell einfache und naheliegende Möglichkeit, jedoch nicht unbedingt die effizienteste. Da aber Effizienz bezüglich aller Ressourcen (Speicher, Rechenaufwand und Energie) sowohl beim Berechnen und Vergleichen als auch bei der Übermittlung gefordert ist, ist im Allgemeinen ein Abwägen zwischen möglichst kompaktem Nachrichtenformat und einer mit minimalem Aufwand in eine einfach zu verarbeitende Form zu bringenden Darstellung nötig, da sich diese beiden Forderungen oft nicht zusammen verwirklichen lassen. Einige Möglichkeiten, die wir uns vor unserer Wahl überlegt haben, möchten wir kurz aufzählen:

Offsetformat: Da die beiden Zeitwerte zweier zusammengehöriger Schranken sich meist nur wenig unterscheiden (dies ist ja gerade das Ziel der Algorithmen: minimale Unsicherheit), lässt sich die Zeitinformation komprimieren, indem eine Schranke in voller Auflösung und die zweite als Offset zur ersten versandt wird. Um eine einigermassen sinnvolle Unsicherheit noch darstellen zu können, sind jedoch je nach vermuteter Kommunikationshäufigkeit und Drift ein bis drei Byte nötig (Mit einem Byte lassen sich 255 Millisekunden, mit zwei ungefähr eine Minute und mit drei bereits vier Stunden darstellen; vgl. auch Abschnitt 3.3.4). Dies führt dazu, dass je nach gewählter

[24] Das ist möglich, da der Wert in Kenntnis der Grösse des Datentyps (`u_long`, 32 Bit) errechnet werden kann und diese Berechnung immer zu einem Zeitpunkt $h > 0$ geschieht, so dass der Timerwert (`u_long`) selbst dann darstellbar ist, wenn `NUT_CPU_FREQUENCY` gesetzt ist und demnach `ticks_left` jede Millisekunde dekrementiert wird.

[25] Die geübte Atemtechnik erfahrener Männerchöre möge hier als anschauliches Beispiel dienen.

Zeitstempelauflösung und -länge eventuell gar keine substantiellen Einsparungen zu erreichen sind, die den Mehraufwand an Berechnungen rechtfertigen würden. Denn zum arithmetischen Vergleich der Schranken müssen beide in voller Form vorhanden sein, was in unserem Fall bereits wieder 64-Bit-Additionen und -Subtraktionen erfordert hätte. Die Berechnung der Schranken (die Prozedur `updateBounds` aus 2.2.2 also) kann immerhin im Offset-Format geschehen. Die Berechnungsvorschrift wird dann beispielsweise so angepasst:

$$T^l(h) = T(h); \qquad T^u(h) = \Delta T(h) + T(h)$$

$$T(h_0 + \Delta h) = T(h_0) + \Delta h/(1 + \rho_{max})$$

$$\Delta T(h_0 + \Delta h) = \Delta T(h_0) + \Delta h/(1 - \rho_{max}) - \Delta h/(1 + \rho_{max})$$

Ein weiteres Problem ist die Frage, wie ein Knoten reagieren soll, wenn seine Unsicherheit grösser als mit dem Offset darstellbar wird. Da er dann nicht einmal mehr interne Synchronisation mit seinen Nachbarn machen kann, muss er im schlechtesten Fall warten, bis neue Information von einem Ankerknoten bis zu ihm propagiert.

Faktorformat: Der eben vorgeschlagene Offset muss nicht bloss ein Wert sein, der einfach zu einer Schranke hinzuaddiert wird; auch ein Faktor kann die Berechnung der zweiten aus der ersten Schranke erlauben. Da auf dem in dieser Arbeit verwendeten Mikrokontroller aber keine Hardware-Divisionen oder Fliesskomma-Operationen möglich sind, wird darauf nicht weiter eingegangen.

Nichtlinearer Offset: Ein Offset kann Information über die zweite Schranke auch komplexer kodieren. Die Grundidee ist, den Offset so zu gebrauchen, dass er sowohl feine Auflösung im tiefen Bereich besitzt als auch sehr grosse Werte noch darstellen kann. Hier ist klar, dass unter Umständen der Rechenaufwand massiv steigen kann, wenn zum Beispiel quadratische oder logarithmische Berechnungen anfallen. Es sind aber einfacher zu berechnende Formate vorstellbar, bei denen zum Beispiel der Offset N Bit $(\beta_1, \beta_2, \ldots, \beta_N)$ weit ist, wobei die ersten k Bit direkt einen binären Wert darstellen und die $(N - k)$ Bit $(\beta_{k+1}, \ldots, \beta_N)$ die Zahl $\sum_{i=k+1}^{N} \beta_i * 2^{i+(i-k)}$ ergeben. Damit liessen sich Zahlen bis $2^{N+(N-k)}$ darstellen, und die Berechnungen wären mit Rightshifts leicht zu implementieren. Dennoch fällt auch wieder negativ ins Gewicht, dass für Vergleich und Berechnung der Schranken die Aufrechnung auf volle Auflösung notwendig ist.

Präfixformat: Es ist nicht nötig, alle hohen Bits immer zu übertragen. Diese und andere Überlegungen führen zum Entwurf eines möglichen ausgereifteren Synchronisationsprotokoll, das nun skizziert wird.

3.5.4 Erweitertes Synchronisationsprotokoll

- *Zeitformat.* Das Versenden eines Zeitstempels, welcher immer die seit 1. Januar 1970 vergangene Zeit in Millisekunden angibt, ist nicht nötig. Da der vordere Teil der Schranken - nachfolgend Präfix genannt - über lange Zeit gleich bleibt, würde es reichen, nur jeweils den hinteren Teil zu verschicken.[26] Zusätzlich könnten die Berechnungen, welche zur Synchronisation nötig sind, auch nur mit dem hinteren Teil ausgeführt werden, um weitere `long long` Operationen zu vermeiden.

- *Selbständigkeit.* Ein BTnode soll auch selbst Präfix und Zeitschranken anfordern können, falls diese Daten ihm nicht bekannt sind oder die Unsicherheit über die Zeit zu gross ist.

- *Adaptive Synchronisationhäufigkeit.* Werden mit einem bestimmten BTnode kontinuierlich Daten ausgetauscht oder ist die Unsicherheit genau genug, so ist es nicht nötig, jedem Paket Informationen über die Zeit mitzugeben. Es sollen auch Pakete versandt werden können, welche keinen Zeitstempel enthalten.

- *Variable Paketlängen* sind eine Voraussetzung für den vorhergehenden Punkt.

Diese Punkte könnten durch Einführung eines erweiterten Synchronisationsprotokolls, welches verschiedene Pakettypen vorsieht, umgesetzt werden. Die Synchronisation wird nicht mehr nur durch implizite, sondern bei Bedarf auch durch explizite Kommunikation realisiert. Das erste Byte des Synchronisations-Headers soll dabei zur Unterscheidung der Paketarten dienen. Folgende Pakettypen müssten implementiert werden:[27]

- `SYNC_NO_TSTAMP`. Das Paket enthält keinen (gültigen) Zeitstempel. Der Synchronisations-Header ist demnach nur ein Byte lang.

[26] Analogie aus dem Alltag: Wird man nach der Zeit gefragt, nennt man nur die Uhrzeit (Stunden und Minuten) und nicht auch noch das Datum, weil klar ist, welches Datum gemeint ist.

[27] Die Synchronisations-Informationen werden wie bis anhin in ACL-Datenpakete integriert. Die Pakettypen bezeichnen also, welche Art von Synchronisations-Paket im ACL-Datenpaket enthalten ist.

- `SYNC_TSTAMP`. Das Paket enthält einen Zeitstempel, welcher je nach Algorithmus unterschiedlich lang sein kann. Der Zeitstempel besteht nur aus dem hinteren Teil der aktuellen Zeit.

- `SYNC_TSTAMP_REQUEST`. Das Paket enthält wie im letzten Punkt einen Zeitstempel, fordert aber zusätzlich eine Antwort an. Bei zu grossem Unsicherheitsintervall kann so ein Zeitstempel angefordert werden. Die implizite Kommunikation wird durch explizite Zweiwegkommunikation ergänzt.

- `SYNC_PREFIX`. Dieses Paket enthält den vorderen Teil der Zeit. Solche Pakete werden nur als Antwort auf `SYNC_PREFIX_REQUEST` und `SYNC_REQUEST` versandt.

- `SYNC_REQUEST`. Der sendende BTnode hat noch keine Zeitinformationen und fordert sowohl einen Zeitstempel wie auch ein Präfix an.

- `SYNC_PREFIX_REQUEST`. Mit diesem Paket fordert ein BTnode ein Präfix an. Dies kann der Fall sein, wenn der BTnode ein `SYNC_TSTAMP` Paket erhält und selbst das Präfix noch nicht kennt. Der BTnode verschickt diese Anfrage dann an den Sender von `SYNC_TSTAMP`, welcher wiederum mit `SYNC_PREFIX` antworten sollte. Sobald ein BTnode das Präfix kennt, darf auch er `SYNC_TSTAMP` Pakete versenden, vorher sollte er jeweils nur Pakete vom Typ `SYNC_NO_TSTAMP`, `SYNC_REQUEST` oder `SYNC_PREFIX_REQUEST` versenden.

Zu beachten ist, dass das Präfix ungültig wird und inkrementiert werden muss, wenn eine Schranke ihren Wertbereich überschreitet. Obere und untere Schranke können somit für kurze Zeit – je nach Unsicherheitsintervall – auch unterschiedliche Präfixe haben. Es wird davon ausgegangen, dass die Abweichung der verschiedenen Zeiten auf den BTnodes klein ist im Vergleich zur Grössenordnung der Präfixe, so dass diese immer eindeutig sind und unter den BTnodes nicht neu kommuniziert werden müssen.

3.5.5 Explizite Synchronisation

Die von uns implementierte implizite Synchronisation könnte bei Bedarf um explizite Synchronisation ergänzt werden. Denkbar ist die Einführung einer Schicht, welche über den Synchronisationsfunktionen liegt und den Status der Synchronisation überwacht. Dazu gehört neben der Intervallgrösse auch die Anzahl aufgrund zu grosser “round-trip time” verworfener Zeitstempel pro Nachbar. Diese Funktionsschicht stellt bei zu grossem Unsicherheitsintervall

den Erhalt von Zeitinformation sicher, indem es diese von allen erreichbaren BTnodes anfordert. Werden zuviele Zeitstempel verworfen, so soll mit einzelnen BTnodes wiederholt kommuniziert werden, bis die Schranken akzeptiert werden.

Diese zusätzliche Funktionalität soll aber von der Anwendung ein- und ausschaltbar sein, falls sie von sich aus schon oft genug kommuniziert oder für längere Zeit keine Kommunikation durchführen will. In letzterem Fall wäre es dann möglich, dass die Uhrensynchronisation weiter durch explizite Synchronisation realisiert würde, oder dass man ganz auf weitere Kommunikation für die Synchronisationsfunktionalität verzichtet und das Bluetooth-Modul ausschaltet.

3.5.6 Nice mode

Auch ein kleines "implizites" Protokoll, das sozusagen auf die Höflichkeit der Knoten setzt und deshalb *nice mode* genannt werden mag, könnte die Synchronisationsqualität verbessern: Knoten würden, wenn sie ungültige Zeitinformation von einem Nachbarknoten erhalten, diesem mit einem Paket antworten, da sie schliessen könnten, dass jener Knoten gern Zeitinformation hätte. Dafür wären keine Änderungen am Schrankenformat nötig; auch müssten nicht alle Knoten diese Vereinbarung befolgen (deshalb die Bezeichnung).

3.5.7 Weitere Highlevel-Funktionen

An der Anwendungsschnittstelle sind einige weitere Funktionen denkbar; die Anwendung der Synchronisationsfunktionalität wird zeigen, welche von ihnen auch wünschenswert sind. Einige Vorschläge:

Prognose-Funktion: Eine Hauptanwendung der Synchronisation sollen ja garantierte Rendezvous zwecks Minimierung des Kommunikationsaufwandes sein. Dafür sollte die Funktion `get_updated_bounds`, die dies eigentlich schon macht, um eine "Hüllenfunktion" ähnlich `rt_get_bounds` erweitert werden, die als Argument vielleicht auch UNIX-Zeitstempel akzeptiert (die zum Beispiel mit den Standard-C-Funktionen aus `time.h` aus einem Datumsstring erzeugt worden sind) und zwei Schranken lokaler Zeit zurückliefert, zwischen denen der gewünschte Zeitpunkt in der tatsächlichen Zeit liegt.

WatchQuality-Funktion: Mit den Timerfunktionen aus Nut/OS ([11]) liesse sich eine Funktion verwirklichen, die ein Callback ausführt, sobald eine im Argument spezifizierte maximale Unsicherheit überschritten wird. Da

jeder Knoten in Kenntnis seiner maximalen Drift den Zeitpunkt voraussagen kann, zu dem dies geschieht, kann einfach ein Timer mit dem entsprechenden Callback gestartet werden. Der globale `struct sync_globals` wird um die Eigenschaften `u_int maximum_uncertainty` und `NUTTIMERINFO * qualtimer` erweitert, welche dazu da sind, im Fall einer Kommunikation (welche die Schranken verbessern und deshalb die Prognose falsifizieren kann) den Timer zu stoppen, eine neue Prognose zu machen und den Timer wieder zu starten. In der Funktion `sync_sync_bounds` muss dann jedesmal abgefragt werden, ob `qualtimer` auf einen gültigen Timer zeigt und allenfalls die Prognose wie erwähnt aufgefrischt werden.

Bessere Schätzung: Vielleicht wäre die Möglichkeit, eine bessere skalare Schätzung der Referenzzeit erhalten zu können, wünschenswert. Neben den erwähnten Varianten, entweder das arithmetische Mittel oder einfach die untere der beiden Schranken zu nehmen, liessen sich mit aufwändigeren Ansätzen bestimmt bessere Schätzungen erzielen, die sowohl kleinen mittleren Fehler als auch Monotonität bieten. Insbesondere aus anderen, nicht intervall-basierten Synchronisationsalgorithmen bekannte Berechnungsweisen wie lineare Regression oder der “convex hull”-Ansatz, die zudem *online* ausgeführt werden können, wären zu prüfen (vgl. dazu auch [4]).

Kapitel 4

Messungen

Dieses Kapitel beschreibt die durchgeführten Messungen und deren Resultate. Zuerst wird kurz auf die Grundlagen unserer Messungen eingegangen, anschliessend werden einige vorbereitende Messungen beschrieben, die zum Einstellen von Parametern nötig waren, und zuletzt werden die eigentlichen Tests der beiden Algorithmen in unserer Versuchsanordnung erläutert.

4.1 Grundlagen

4.1.1 Referenzzeit

Bei den Messungen wurde darauf geachtet, dass der Ankerknoten die Zeit vom selben PC bezog, mit welchem später die Schranken von den einzelnen BTnodes abgerufen wurden bzw. welcher als Taktgeber für die Speicherzeitpunkte der Schranken diente. Dadurch musste nicht auch noch die Zeit zwischen mehreren PCs synchronisiert werden, was eine weitere Unsicherheit zu den gemessenen Zeiten hinzugefügt hätte. Der PC bezog seine Zeit wiederum über NTP. Es kann also davon ausgegangen werden, dass die Softwareclock des PCs näherungsweise eine Drift von 0 hatte.

4.1.2 Parameter

Die Resultate der folgenden Messungen hängen natürlich von einer Unzahl an Parametern ab. Wichtige Rollen spielen Batteriespannung, Prozessorlast, Störungen durch andere Kommunikation im selben Frequenzband, Priorität der einzelnen Threads usw. Im Rahmen dieser Arbeit konnten nur ausgewählte Varianten aller möglichen Parameterkonfigurationen getestet werden. Um präzisere Aussagen über den Einfluss der Parameter beziehungsweise deren optimale Werte machen zu können, müssten die von diesen Parametern

abhängigen Eigenschaften der BTnodes und der implementierten Synchronisationsfunktionen unter allen möglichen Parameterkonfigurationen genauer untersucht werden. Die im Rahmen dieser Arbeit durchgeführten Experimente berücksichtigten bloss einige wenige, uns sinnvoll und relevant erscheinende Parameterkonfigurationen, die in Abschnitt 4.3.2 näher beschrieben sind. Die damit erhaltenen Resultate können als grober Anhaltspunkt dienen.

4.1.3 Messgrössen

Als Kriterien zur Beurteilung der Algorithmen wählten wir einerseits die Grösse des Unsicherheitsintervalls (UN), um diese auch mit den Simulationen aus [1] vergleichen zu können. Zusätzlich wollten wir aber die Genauigkeit unserer Implementierung testen und somit auch die Abweichung der BTnode-Zeit – gebildet aus dem Mittelwert der oberen und unteren Schranke – von der PC-Zeit messen, was in [4] als *Synchronization Error (SE)* bezeichnet wird. Als weiteres Mass betrachteten wir die mittlere maximale Zeitabweichung zwischen allen BTnodes, *Instantaneous Precision (IP)* genannt. Weiter wurde überprüft, in wieviel Prozent der Messungen die Echtzeit effektiv zwischen den Schranken lag (G), um zu sehen ob die Implementierung korrekt funktioniert. Alle Messgrössen wurden nur zu Zeiten, bei welchen der BTnode im Besitz von gültigen Schranken war, ausgewertet.

4.1.4 Probleme

Eine Reihe von Problemen ergab sich bei den Messungen. Um zeitraubendes Fehlersuchen in zukünftigen Arbeiten zu reduzieren, sollen diese hier kurz erläutert werden.

Beim Auslesen der Daten über die UART-Schnittstelle zwischen PC und BTnode traten immer wieder Störungen auf. Bei einzelnen Abfragen waren die erhaltenen Zahlenwerte offensichtlich falsch, da sie um mehrere Grössenordnungen von den übrigen Messungen abwichen. Es stellte sich heraus, dass in den jeweils übertragenen Character-Streams, die von uns ja eigentlich als Bit-Streams verwendet wurden, Zeichen vorkamen, welche vom PC als Steuerzeichen (*XON* und *XOFF*) interpretiert wurden und deshalb die ausgelesenen Werte verfälschten[1]. Der Auslesemechanismus wurde deshalb abgeändert, indem der Bitstrom sozusagen “4-Bit-Byte-kodiert” wurde: Ein

[1] Wir hatten vor der Implementierung der Datenübertragung dieser Möglichkeit vermeintlich Rechnung getragen, indem wir abklärten, ob die UART eventuell gewisse Characters nicht übertragen kann, was unseres Wissens nicht der Fall ist. Die UART betreibt keine Flusskontrolle, die Linux-Devicetreiber am anderen Ende der Schnittstelle jedoch schon.

Byte auf dem BTnode wurde zur Übertragung in 2 Bytes geteilt, wobei bei den übertragenen Characters die ersten 4 Bit so gesetzt wurden, dass sich in Kombination mit den zweiten 4 Bit – vom zu übertragenden Wert herrührend – immer darstellbare Characters ergaben und keine Steuerzeichen mehr auftreten konnten.

Weiter wurde beobachtet, dass bei der Kommunikation zwischen Ankerknoten und PC eine Abfrage des gesamten Zeitstempels auf einmal mittels `_read()` nicht korrekt funktionierte. Einzelne Zeichen wurden zum Teil zu Null gesetzt oder gingen verloren, was Flusskontrolle auch auf Seite der Nut/OS-Treiber vermuten lässt. Ein zeichenweises Lesen in einer `for()`-Schleife zeigte diesen Effekt jedoch nicht.

Teilweise schwankten die ausgelesenen Werte ziemlich stark. Ein periodisches Muster war oft erkennbar. Die Schwankungen waren zudem je nach verwendetem Adapter unterschiedlich stark. Die Abweichung der Schranken von der Referenzzeit stieg meist stetig, hatte aber zwischendurch markante Verschiebungen nach unten. Da dies für beide Schranken gleichzeitig galt, gingen wir davon aus, dass deren Berechnung auf den BTnodes korrekt erfolgte, das Auslesen über die UART-Schnittstelle und die Auswertung auf dem PC aber starken Schwankungen unterlag. Auch eine Variante, bei der die BTnodes periodisch ihre Schranken an den PC schickten, der in einer Polling-Schleife war und jeweils die Ankunftszeitpunkte der Daten als Referenzzeit nahm, behob das Problem nicht. Schliesslich beschlossen wir, die Zeit lokal auf den BTnodes zu bestimmten Zeitpunkten zu speichern, wobei der PC als Taktgeber fungierte[2]. Die Auswertung dieser Resultate bestätigte unsere Vermutung, da nun praktisch keine Schwankungen mehr erkennbar waren.

4.2 Erste Messungen

4.2.1 Nachrichtenverzögerung

Um die Nachrichtenverzögerung zwischen zwei BTnodes zu messen, wurden beide BTnodes an einen PC angeschlossen, um den Messvorgang zu steuern. Es wurde darauf geachtet, dass die Verzögerung auf möglichst tiefer Ebene gemessen wurde. Der Messvorgang lief folgendermassen ab:

1. Der PC veranlasst den ersten BTnode, ein Paket an den zweiten BTnode zu senden.

[2] Über den Parallelport des PC wurde mit einem (galvanisch) verzweigten Kabel auf allen BTnodes der Testanordnung gleichzeitig während einiger Mikrosekunden ein externer Interrupt getriggert, welcher höhere Priorität als alle Timer und UARTs besitzt.

2. Der erste BTnode misst die Anzahl seit Programmstart vergangener Millisekunden zum Zeitpunkt $send_1$, gerade nachdem die Synchronisationsfunktionen aufgerufen wurden und kurz bevor die Daten des zu versendenden Pakets auf die UART-Schnittstelle zum Bluetooth-Modul geschrieben werden.

3. Der zweite BTnode speichert den Wert der lokalen Uhr zum Zeitpunkt $rcvd_1$ zwischen dem Lesen des Zeitstempels des Pakets von der UART-Schnittstelle und dem Aufruf der Synchronisationsfunktionen und verschickt ein Antwortpaket, wobei er den Zeitpunkt $send_2$ vor dem Schreibvorgang auf die UART-Schnittstelle speichert.

4. Der erste BTnode wiederum speichert beim Empfang des Antwortpakets den Wert der lokalen Uhr zur Zeit $rcvd_2$ nach dem Lesen des Zeitstempels.

5. Der PC ruft die vier gespeicherten Messwerte von den beiden BTnodes ab und berechnet die "round-trip"-Verzögerung t_{RT} ohne Verarbeitungszeit auf dem zweiten BTnode:

$$t_{RT} = (rcvd_1 - send_1) - (send_2 - rcvd_2)$$

6. Nach einer kurzen Wartezeit wird mit Punkt 1 weitergefahren und eine weitere Messung durchgeführt.

Für alle Konfigurationen wurden jeweils 200 Messungen durchgeführt. Die Nachrichtenverzögerung für IM und BP-ISA in Abhängigkeit der Grösse der mitgeführten Payload zeigt das Histogramm in Abbildung 4.1.

Folgende Tabelle fasst die ermittelten Mittelwerte über die "round-trip"-Verzögerung t_{RT} zusammen.

effektive Payload (ohne Zeitstempel)	10 Bytes	50 Bytes	90 Bytes
$\bar{t}_{RT}$(IM)	17.165 ms	20.930 ms	25.255 ms
$\bar{t}_{RT}$(BP-ISA)	22.045 ms	30.800 ms	40.470 ms

Im Implementierungs-Ansatz "Verzögerungs-Statistik" wurde anhand von diesen Werten die Nachrichtenverzögerung eines empfangenen Pakets in Abhängigkeit der Payload-Länge geschätzt.

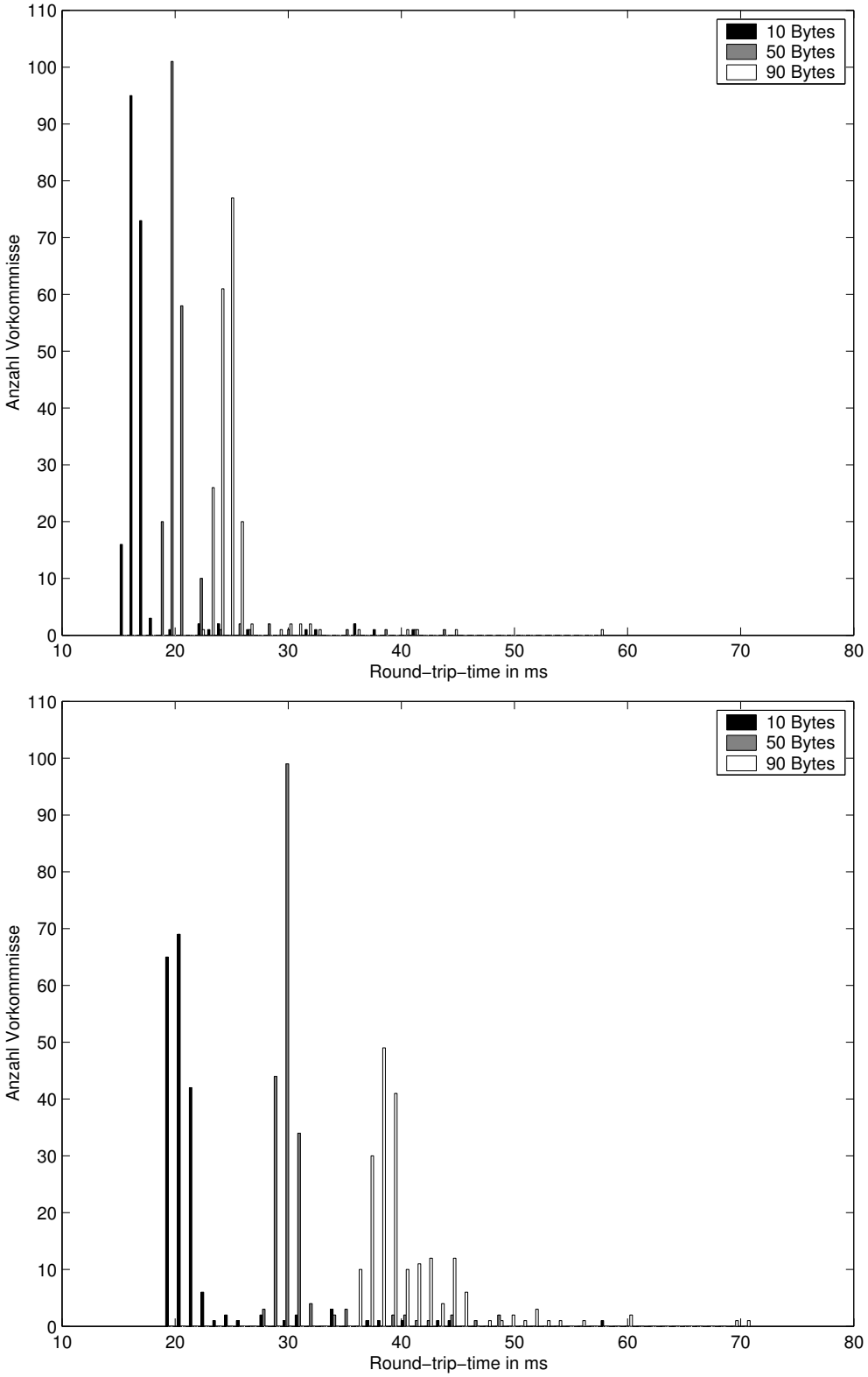

Abbildung 4.1: Nachrichtenverzögerung bei IM(oben) und BP-ISA für eine effektive Payload (ohne Zeitstempel) von 10, 50 und 90 Bytes. Für jede Länge wurden 200 Messungen gemittelt.

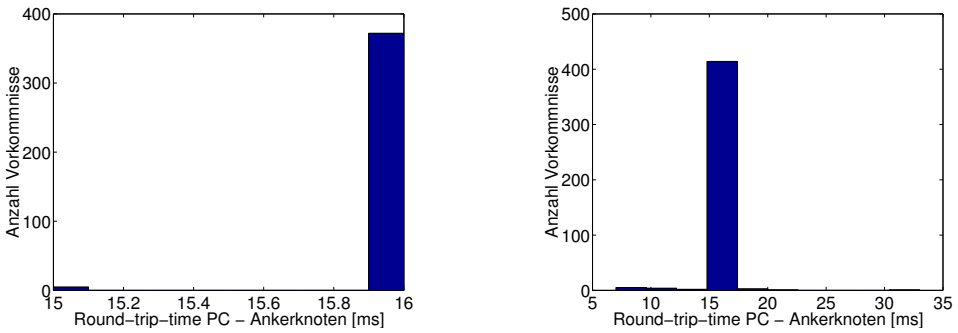

Abbildung 4.2: Verzögerung zwischen PC und Ankerknoten bei IM (links) und BP-ISA.

4.2.2 Kommunikation zwischen Ankerknoten und PC

Um zu bestimmen, wieviel Zeit zwischen dem Generieren eines Zeitstempels auf dem PC und dem Absenden auf dem Ankerknoten vergeht, wurde in `tstampserver` die Zeit gespeichert, zu welcher der Ankerknoten die Bestätigung über den Erhalt des Zeitstempels an den PC zurückschickte. Diese Zeit wurde dann mit der in dem Zeitstempel enthaltenen Zeit verglichen und die Differenz gebildet.

Die Ergebnisse sind in Abbildung 4.2 als Histogramm dargestellt. Für IM ergab sich in 98.67% der 377 Messungen, für BP-ISA in 94.65% der 430 Messungen eine Verzögerung von 16 ms.

4.2.3 Kette von BTnodes

Die Synchronisationsqualität über mehrere Hops wurde mit dem Programm `sync-test` getestet, welches so konfiguriert war, dass ein BTnode nur jeweils mit seinem unmittelbaren Nachbarn auf der einen Seite kommunizieren durfte, wobei die BTnodes in einer Kette angeordnet waren. Die Zeitinformation wurde also von BTnode zu BTnode weitergereicht. Die Auswertung dieser Messungen ergab, dass unsere erste Implementierung ("Verzögerungs-Statistik") nicht sehr gut funktionierte. Pakete mit neuen Zeitschranken wurden teilweise so stark verzögert, dass die Echtzeit nicht mehr zwischen den Schranken lag. Damit war die Bedingung 2.2 nicht mehr erfüllt und somit die Korrektheit des Algorithmus nicht mehr gegeben. Aus diesem Grund wurde dann auch beschlossen, unsere Implementierung um die Online-Messung der "round-trip time" jedes einzelnen Pakets zu erweitern, wie in Abschnitt 3.3.7 erläutert wurde.

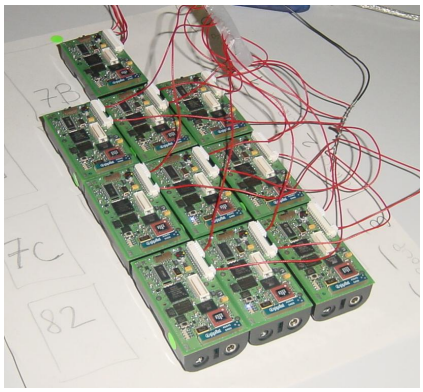

Abbildung 4.3: Testaufbau bei den ausführlichen Messungen.

4.3 Ausführliche Messung

4.3.1 Messaufbau

Die Algorithmen wurden in einem Netz von 10 BTnodes getestet (siehe Abbildung 4.3), wovon einer als Ankerknoten diente. Das Netz wurde in drei Schichten mit gleicher minimaler Anzahl Hops zum Ankerknoten eingeteilt, wobei den BTnodes nur die Kommunikation innerhalb ihrer eigenen Schicht beziehungsweise – falls vorhanden – mit der darüber und darunter liegenden Schicht erlaubt war. Auf den BTnodes wurde `AUTO_COMMUNICATE` und `AUTO_ANSWER2` gesetzt, um eine kommunizierende Anwendung zu simulieren. Auf den 9 Sensorknoten wurde die Grösse der Schranken während des Messvorgangs mittels Hardware-Interrupts periodisch gespeichert, wie es in Abschnitt 4.1.4 geschildert wurde. Dabei diente derselbe PC, welcher als "timestamp-server" eingesetzt wurde, auch als Auslöser dieser Interrupts, um eine einzige, konsistente Referenzzeit im Versuchsaufbau zu haben. Nach der gewünschten Anzahl Iterationen wurden die gespeicherten Schranken von den BTnodes ausgelesen und mit den auf dem PC gespeicherten Referenzzeitpunkten (den Zeitpunkten, zu denen die Interrupts erfolgt waren) kombiniert, um dann die jeweiligen Abweichungen der Schranken aller Knoten gegenüber der Referenzzeit zu allen Messzeitpunkten zu berechnen.

4.3.2 Auswertung

Um die Messergebnisse auszuwerten, wurde ein MATLAB-Skript geschrieben, mit welchem die weiter oben definierten Messgrössen pro Schicht und für das ganze Netzwerk berechnet sowie Plots erstellt werden konnten.

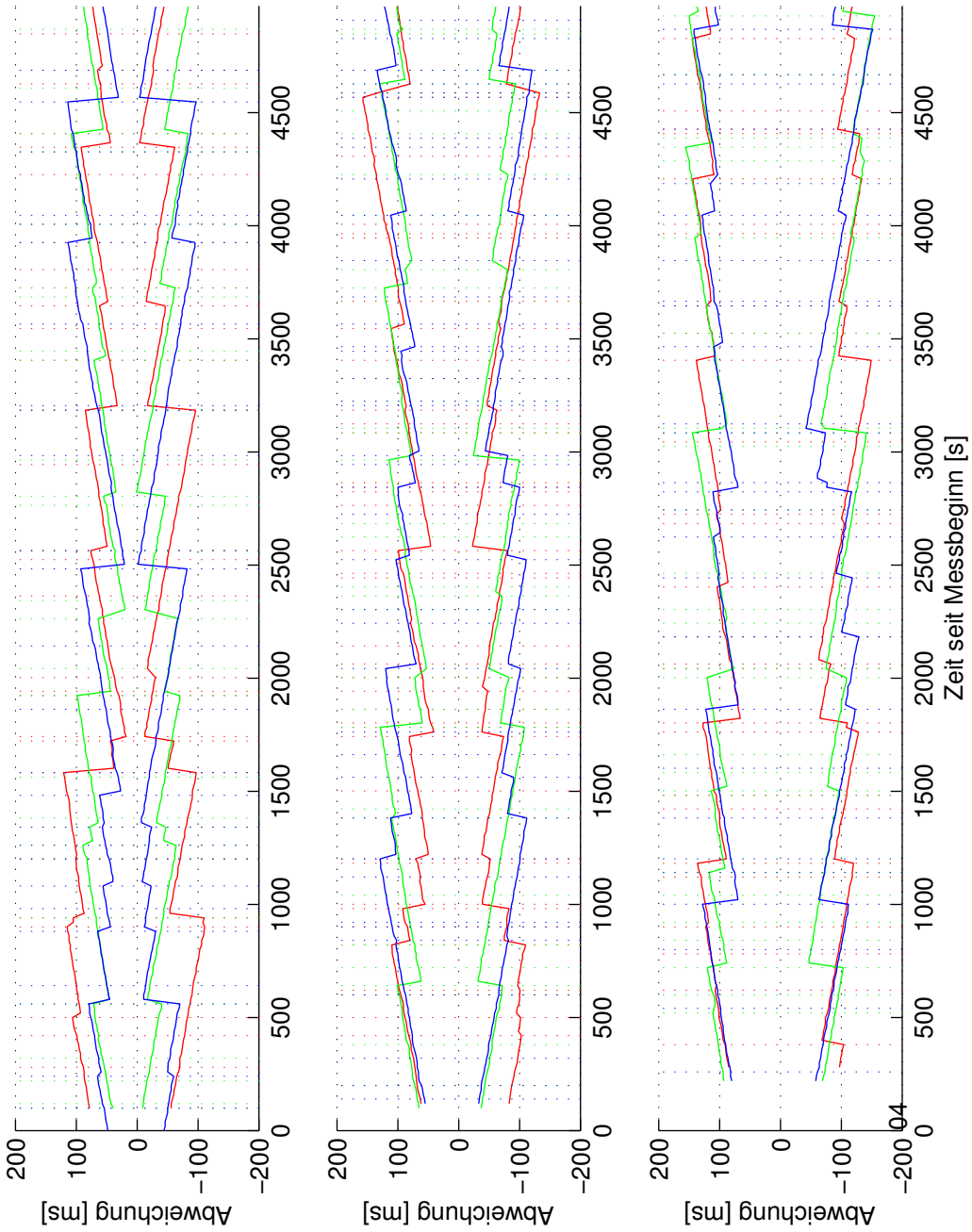

Abbildung 4.4: Abweichung der Schranken der 9 Sensorknoten von der Referenzzeit für eine Messung mit BP-ISA, aufgeteilt nach den 3 Schichten. Das oberste Diagramm zeigt die Abweichung für die 3 BTnodes der Schicht, welche am nächsten zum Ankerknoten ist.

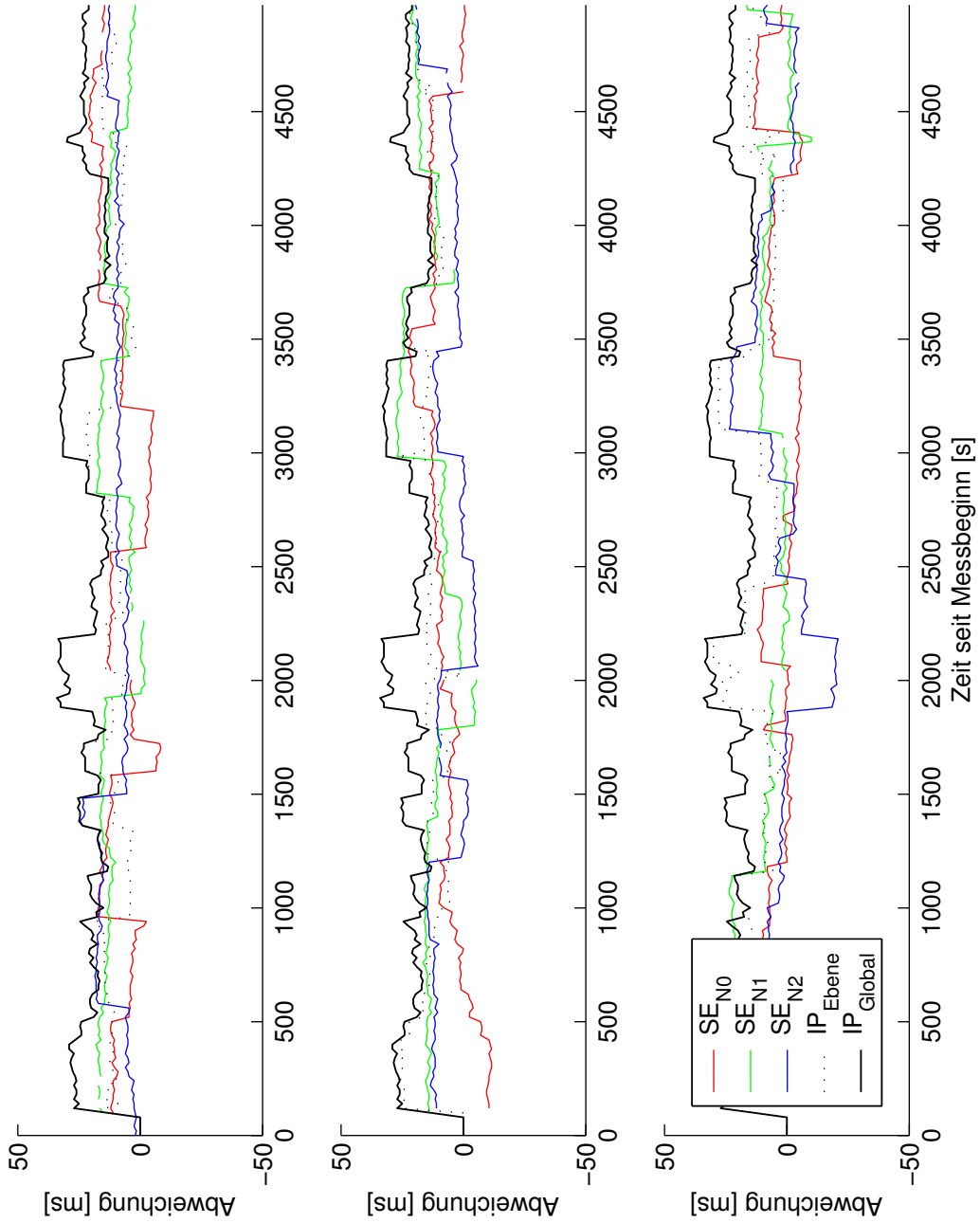

Abbildung 4.5: *Synchronization Error (SE)* und *Instantaneous Precision (IP)* der 9 Sensorknoten für eine Messung mit BP-ISA, aufgeteilt nach den 3 Schichten. Das oberste Diagramm zeigt die Messwerte für die 3 BTnodes der Schicht, welche am nächsten zum Ankerknoten ist.

Eine typische Messung

Ein Beispiel einer Messung zeigt Abbildung 4.4. Hierbei wurde der Algorithmus BP-ISA verwendet. Die auf den BTnodes laufende Anwendung versandte Pakete mit einer zufälligen Länge von 1 bis 50 Bytes im Abstand von 3 bis 7 Minuten. Geantwortet wurde mit einer fixen Paketlänge von 10 Bytes. Als minimale Zeit für die Nachrichtenverzögerung (nur in eine Richtung) wurden 12 Millisekunden angenommen. Bei Nachrichten mit einer "round-trip time" grösser als 140 Millisekunden wurde die darin enthaltene Zeitinformation ignoriert. Während zirka 80 Minuten wurden die Schranken auf den BTnodes alle 20 Sekunden gespeichert. Im obersten Plot sind die Schranken der 3 BTnodes der Ebene direkt unter dem Ankerknoten mit den Farben rot, grün und blau dargestellt, im mittleren und unteren Plot die Schranken der BTnodes aus Ebene 2 bzw. 3. Die Plots beginnen jeweils zu dem Zeitpunkt, bei welchem der erste BTnode gültige Schranken erhält. Ebenfalls eingezeichnet – mit einer vertikalen Linie in der Farbe des jeweiligen BTnodes – sind die Zeitpunkte, zu welchen sich der `last_update` Wert der aktuellen Schranken geändert hat. Für IM ist dies immer der Fall, wenn ein Paket empfangen wird, für BP-ISA beim Senden und Empfangen.

In der Abbildung ist erkennbar, wie lange es dauert, bis die einzelnen BTnodes gültige Schranken besitzen. Weiter lässt sich beobachten, dass das Unsicherheitsintervall von Ebene zu Ebene zunimmt. Dies entspricht den Erwartungen: Je mehr Hops zwischen dem BTnode und dem Ankerknoten liegen, desto schlechter sind die Informationen über die Zeit, welche ihn erreichen.

Weiter gibt Abbildung 4.5 einen Überblick über die Synchronisationsqualität im Verlauf der Zeit. Es wird der SE für jeden einzelnen Knoten, die IP für alle Knoten aus einer Ebene und die IP für alle Knoten im Netz gezeigt.

In Tabelle 4.1 ist das Ergebnis der Auswertung in MATLAB zu sehen. Für jede Ebene werden die Werte zuerst für die 3 Knoten einzeln angegeben, danach für die gesamte Ebene. In der letzten Zeile stehen die Werte, welche für das gesamte Netz berechnet wurden. In der dritten Spalte findet sich der mittlere Synchronisationsfehler. Es folgt die durchschnittliche Grösse des Unsicherheitsintervalls sowie das Verhältnis der Anzahl Messungen, bei welchen die Referenzzeit zwischen den Schranken lag, zu der gesamten Anzahl Messungen. Für jede Ebene bzw. für das gesamte Netz wird zusätzlich der mittlere Wert der IP angegeben. Die letzte Reihe zeigt, wieviele Messungen berücksichtigt wurden.

Es ist zu sehen, dass die Abweichung zwischen zwei BTnode-Zeiten im Schnitt 10 ms und maximal – wie in Abbildung 4.5 ersichtlich – 34 ms beträgt. Die Abweichung von der Referenzzeit beträgt im Schnitt 7 ms und die

Ebene	Knoten	$\overline{SE}[ms]$	$\overline{UN}[ms]$	$\overline{G}$	$\overline{IP}[ms]$	# Messwerte
1	0	9.090	124.222	1.000		239
1	1	10.508	109.767	0.996		240
1	2	9.807	108.602	1.000		249
1	alle	9.803	114.114	0.999	1.418	728
2	0	7.419	159.934	1.000		241
2	1	13.562	158.389	1.000		242
2	2	6.246	177.250	1.000		240
2	alle	9.086	165.165	1.000	7.316	723
3	0	3.511	216.440	1.000		234
3	1	6.918	214.431	1.000		232
3	2	3.617	197.699	1.000		236
3	alle	4.672	209.476	1.000	3.407	702
alle	alle	7.889	162.351	1.000	10.051	2153

Tabelle 4.1: Messgrössen für verschiedene Knoten, Ebenen und für das gesamte Netz (gerundet auf 3 Nachkommastellen).

durchschnittliche Grösse des Unsicherheitsintervalles ist 162 ms. Diese Werte waren von der Grössenordnung typisch für alle Messungen.

Herabsetzung der maximal akzeptierten RTT

Es zeigte sich, dass eine Herabsetzung der maximal akzeptierten "round-trip time" von 140 ms auf 115 ms bei IM die durchschnittliche Unsicherheit über die Zeit im Schnitt um zirka 20 ms erhöhte.

Durch die Reduktion entstehen zwei Effekte. Einerseits werden die Pakete öfter verworfen und die Intervalle damit auch seltener kombiniert. Andererseits kann die Grösse des Unsicherheitsintervalls nach einem Kommunikationsereignis weniger gross sein bei kleinerer maximaler RTT. In dieser Messung überwiegt also der erste Effekt.

Es wäre die Aufgabe weiterer Arbeiten, den optimalen Wert für die maximal akzeptierte RTT in Abhängigkeit des gewählten Algorithmus durch Messungen genauer zu ermitteln.

Seltene Kommunikation mit Ankerknoten

Weiter wurde ein Experiment durchgeführt, bei welchem der Ankerknoten zunächst kommunizierte, danach aber ausgeschaltet wurde. Die Grösse des

Unsicherheitsintervalles ist in der Folge natürlich sehr stark angestiegen. Zwei Beobachtungen konnten gemacht werden: Erstens blieb die durchschnittliche IP über das ganze Netz konstant. Das Sensornetz blieb demnach intern sehr gut synchronisiert, auch wenn keine neuen Zeitinformationen mit kleinem Unsicherheitsintervall verfügbar waren. Der absolute SE veränderte sich mit 3.3 ppm nur sehr langsam (Veränderung von +5 ms auf -15 ms während 6000 Sekunden). Dieses zweite Ergebnis zeigt, dass die BTnodes mit den korrekten Parametern für den maximalen Drift auch über lange Zeit extern relativ gut synchronisiert bleiben.

Kapitel 5

Schlussfolgerungen

5.1 Beurteilung der Algorithmen und ihrer Implementierung

Im beschränkten zeitlichen Rahmen dieser Arbeit und wegen aller Verzögerungen durch die Probleme (insbesondere mit dem sauberen Auslesen der Zeit beim Messen) ist es uns oft erst relativ spät gelungen, Fehler in den Implementierungen der Algorithmen zu finden. Da auch immer ein gewisser Teil der Messungen aus verschiedenen Gründen scheiterte, liegen deshalb eigentlich zu wenig Daten vor, um aussagekräftige Schlüsse zu ziehen. Dennoch versuchen wir, einige Beobachtungen zu schildern und Erklärungen dafür zu liefern.

5.1.1 Robustheit

Beide Algorithmen weisen dank der einfachen Schnittregel und dem (vom Messen der RTT abgesehen) "zustandslosen" Synchronisationsprotokoll sowie der transparenten Einbettung der Funktionen im Betriebssystem und der Daten in der Payload eine hohe Robustheit auf. Wird ein Knoten neu gestartet oder hat er infolge eines Fehlers vorübergehend ungültige Schranken, so kann er im Normalfall bereits nach einer Kommunikation mit einem Nachbarn wieder im Besitz gültiger Schranken sein. Heikler gestaltet sich dies beim Ausfall eines Ankerknotens (oder des damit verbundenen PC's). Einerseits ist es ratsam, Redundanzen bei der Zeiteinspeisung einzuplanen, andererseits haben die Tests gezeigt, dass bei "internem" Betrieb die Unsicherheit zwar gross wird, die mittlere Abweichung unter den Knoten aber klein bleibt. Aus diesen Gründen können beide Algorithmen als sehr robust und deshalb in dieser Hinsicht als für den Einsatz in verteilten Sensornetzen

geeignet gelten.

5.1.2 Implementierungsaufwand

Der Algorithmus IM besticht in seiner theoretischen Form durch seine Einfachheit; auch BP-ISA ist von ähnlicher Komplexität, jedoch sind umfangreichere Datenstrukturen nötig. In der tatsächlichen Verwirklichung sind aber auch die Datenstrukturen im Falle der Variante "RTT-Messung" praktisch identisch. Nähere Untersuchungen der einzelnen Berechnungszeiten könnten vielleicht genaueren Aufschluss über den Mehraufwand von BP-ISA gegenüber IM geben, der beim Ausführen von `updateMemory` beziehungsweise `update_neighbours` entsteht. Falls dieser sich in vertretbaren Grenzen hält, so können beide Algorithmen als mit vergleichbarem Aufwand implementierbar gelten.

5.1.3 Qualität

Die erzielbare Qualität liegt bei beiden Algorithmen in einem ähnlichen Bereich, und die Grössenordungen der Messwerte stimmen auch mit denjenigen der Softwaresimulationen in [1] überein. Die Unsicherheit nimmt zwar mit zunehmendem Abstand vom Ankerknoten (Pfadlänge) zu, im Fall "RTT-Messung" nicht aber der mittlere Fehler. Die Bedingung 2.2, wonach die tatsächliche Zeit oder Referenzzeit immer zwischen den beiden Schranken liegen muss, ist erfüllt. In unseren Tests war noch kein Vorteil von BP-ISA gegenüber IM erkennbar. Dies liegt vermutlich daran, dass die ausgeführten Tests zu wenig und zu kurz waren. Vermutlich wirkt sich der Informationsgewinn durch Backpath-Berechnung erst aus, wenn die Knoten mehrfach miteinander kommuniziert haben und die Intervalle zwischen den Kommunikationsereignissen gross sind, damit Driftunterschiede wirksam werden können und Verbesserungen auf gespeicherten Schranken, die als $\vec{T}_{previous}$ mitgesendet werden, grösser als die beim Senden erzeugte zusätzliche Unsicherheit sind. Gerade wegen des letzten Punkts könnte sich eine engere Abschätzung der Unsicherheit über die Nachrichtenverzögerung hier als doppelt hilfreich erweisen, da dann nicht nur Schranken $\vec{T}_{new}$ mit geringerer Unsicherheit übertragen werden können, sondern auch die gespeicherten Zeitpunkte $h_M[N_i]$ mit kleinerer Unsicherheit gegeben wären und mehr Information aus der Kombination von gespeicherten ($\vec{T}[N_i]$) und empfangenen Schranken ($\vec{T}_{previous}$) gewonnen werden könnte.

In einigen Tests resultierte mit BP-ISA gar eine grössere Unsicherheit, was erstaunen mag, da formal gezeigt werden kann, dass der Algorithmus

nie schlechter als IM sein kann[1]. Unterschiedlich lange Berechnungszeiten infolge verschiedener Komplexität sollten dabei keine relevante Rolle spielen, da die Berechnungen ausserhalb der Nachrichtenübertragungszeit geschehen. Eine mögliche Erklärung ist, dass die wegen des grösseren Zeitstempels längere effektive Payload eine grössere Schwankung der Unsicherheit über die Nachrichtenverzögerung verursacht. Dies ist darum möglich, weil die *best-case* Werte für die Übertragungsdauer fast identisch mit denen der durchschnittlichen IM-Pakete ist (wie aus der Abbildung 4.1 auf Seite 49 ersichtlich ist), die *worst-case* Werte aber um einige dutzend Millisekunden höher sind. Wir erklären uns dies mit einem stochastischen Argument: Zwei mögliche Szenarien, weshalb überhaupt Nachrichtenverzögerungen in der Grösse von bis zu 200 Millisekunden auftreten, sind *Retransmission* eines Pakets durch das Bluetooth-Modul und Ausführen eines *Threadswitches* in der Nut/OS-Funktion `fflush()`, die von `_bt_hci_send_pkt` beziehungsweise von `_UART_write` aufgerufen wird. Bei längeren Paketen treten beide Fälle mit *höherer Wahrscheinlichkeit* auf. Dies könnte das beobachtete Phänomen erklären.

5.2 Weitere Arbeiten

Folgende weitere Punkte könnten im Zusammenhang mit dieser Arbeit noch umgesetzt werden:

- Es sollten ausführlichere Tests durchgeführt werden, um die Qualität der Implementierung und Synchronisation genauer zu evaluieren und die Unterschiede zwischen den beiden implementierten Algorithmen mit grösserer statistischer Signifikanz beschreiben zu können. Besonders interessant dürfte ein Test mit noch mehr Sensor- und Ankerknoten sein.

- Die Unsicherheit über die Nachrichtenverzögerung sollte unserer Meinung nach noch weiter reduzierbar sein und somit auch eine Verbesserung des Synchronisationsfehlers mit sich bringen. Dazu wäre es nützlich, genau festzustellen, wo die einzelnen Verzögerungen bzw. Schwankungen herrühren, um konkrete Massnahmen gegen deren Einfluss auf die Synchronisationsqualität zu treffen. Hierbei ist fraglich, wie weit eine Verbesserung bei Benutzung des Bluetooth-Moduls noch möglich ist und ob sich das ChipCon-Radiomodul der BTnodes dank direkterem Zugriff auf tiefere Ebenen nicht besser eignen würde.

[1] Vgl. [1]. Dies ist auch intuitiv klar: Aus *zusätzlich* verfügbarer Information kann nie ein Informationsverlust resultieren.

- Weiter könnten die unter Abschnitt 3.5 erwähnten Punkte – das erweiterte Protokoll sowie die zusätzliche Funktionsschicht – umgesetzt werden.
- Da dank dieser Arbeit die Uhrensynchronisation nun den Anwendungen zur Verfügung steht, wäre ein Praxistest mit realen Anwendungen angebracht.

5.3 Erzielte Resultate

Die ersten beiden eingangs in Abschnitt 1.3 gestellten Ziele dürfen als erreicht betrachtet werden: Beide Algorithmen wurden auf BTnodes implementiert und stehen als Dienst in BTnut den Anwendungen zur Verfügung. Das dritte Ziel, Vergleich der Algorithmen untereinander und der praktischen Implementierung mit derjenigen der Softwaresimulation, konnte nur teilweise erreicht werden. Die Zeit reichte nicht für ausreichend häufiges und langes Testen der finalen Versionen der beiden Algorithmusimplementierungen, wobei verschiedene Effekte den Vergleich erschwerten. Immerhin wurde eine brauchbare Qualität mit angemessenem Ressourcenaufwand erreicht, die überdies im Bereich der in der Softwaresimulation errechneten Werte lag.

Literaturverzeichnis

[1] Philipp Blum, Lennart Meier und Lothar Thiele. *Improved Interval-Based Clock Synchronization in Sensor Networks*. IPSN'04, April 2004. http://www.tik.ee.ethz.ch/~meier

[2] Keith Marzullo und Susan Owicki. *Maintaining the time in a distributed System*. Proceedings of the second annual ACM symposium on principles of distributed computing. ACM Press, 1983, pp. 295-305

[3] Mathias Payer. *Implementation of a Bluetooth Stack for BTnodes and Nut/OS*. Studienarbeit am Institute for Pervasive Computing, Distributed Systems Group der ETH Zürich, 2004.

[4] Kay Römer, Philipp Blum und Lennart Meier. *Sensor Networks*, Kapitel Time Synchronization and Calibration in Wireless Sensor Networks. Wiley & Sons, New York, 2005.

[5] Boris Zweimüller. *Uhrensynchronisation in einem Ad-Hoc-Netz von BTnodes*. Studienarbeit SA-2004-06 am Institut für Technische Informatik und Kommunikationsnetze (TIK) der ETH Zürich, 2004.

[6] *Atmel ATmega128*. Datenblatt. http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[7] *Bluetooth Specification Version 1.2*. https://www.bluetooth.org/foundry/adopters/document/Bluetooth_Core_Specification_v1.2

[8] *Epson FC-135/145 SMD Low Frequency Crystal Unit*. Datenblatt. http://www.epson-electronics.de/cgi-bin/panamafe/panama/demand/home.do?nextPage=/demand/catalog/browseCatalog.do&nextPageParams=tabId:1|categoryOid:-8253

[9] *Network Time Protocol (NTP), RFC 958.*
`http://www.faqs.org/rfcs/rfc958.html`
Vgl. auch RFCs 1059 (Version 1), 1119 (Version 2), 1305 (Version 3) sowie 2030 (SNTP Version 4).

[10] *Nut/OS.* `http://www.ethernut.de/en/software.html`

[11] *Nut/OS: Threads, Events and Timers.*
`http://www.ethernut.de/pdf/entet100.pdf`

Anhang A

Quellcode-Auszug

Wie bereits erwähnt, existieren verschiedene Varianten des Quellcodes. Nachfolgend wird die Variante BP-ISA mit “RTT-Messung” auf Sensorknoten gezeigt.

A.1 sync.h

```
1   /**
2   * @file sync.h
3   *
4   * @brief Definition of synchronization functions
5   *
6   */
7  #ifndef SYNC_H_INCLUDED
8  #define SYNC_H_INCLUDED
9
10 #include <sync/globals.h>
11
12 #include <stdio.h>
13 #include <stdlib.h>                              //only for debug printing (we need abs()
       in _print_dec)
14
15 #include <sys/atom.h>
16 #include <sys/timer.h>
17
18 #include <bt/bt_hci_cmds.h>
19 #include <bt/bt_hci_defs.h>
20 #include <bt/bt_hci_api.h>
21
22 #include <led/btn-led.h>
23
24 #if defined(BP_ISA) || defined(SYNC_RTT)
25         #include <sys/heap.h>
26 #endif
27
28  /*****************************************************************************
29  * FUNCTION LIKE PREPROCESSOR MACROS
30  *****************************************************************************/
31
32 /**
33  * Returns the current value of the  software clock in milliseconds (with 32-bit
       resolution (u_long) and 1-ms granularity)
34  */
35 #ifdef NUT_CPU_FREQ
36         #define MILLIS NutGetMillis()
37 #else
38         #define MILLIS NutGetRealMillis()
39 #endif
40
41 /**
```

```
42   * Actual value of the software clock in milliseconds
43   */
44  #define RT_NOW MILLIS
45
46  /**
47   * Average of the lower 4 byte of two 64-bit (long long) values, rounded to 32-bit
48   */
49  #define RT_COARSE_AVERAGE(a, b) (((u_long) (a) + (u_long) (b)) >> 1)
50
51  /**
52   * Copy a bluetooth address
53   */
54  #define BD_ADDR_CPY(D, S)         {(D)[0]=(S)[0]; (D)[1]=(S)[1]; (D)[2]=(S)[2]; (D)[3]=(S)
        [3]; (D)[4]=(S)[4]; (D)[5]=(S)[5];}
55
56
57   /*****************************************************************************
58   * TUNEABLE DCC PARAMETER
59   *****************************************************************************/
60
61  /**
62   * Maximal drift of clock rate (bounded-drift model) in ppm, scaled by 1'000'000
63   */
64  #define DRIFT_CONSTRAINT 65
65
66  /**
67   * Adjusts the ratio of maximum to minimum drift: For DRIFT_ADJUST = 0.5, the absolutes
         of both maximum and minimum drift
68   * are equal; the bigger this value is, the faster the upper bound grows.
69   * Thus one can take into account knowledge about the average of the clock speed.
70   */
71  #define DRIFT_ADJUST 0.75
72
73  /**
74   * Precalculated factor for calculating current bounds
75   */
76  //precalculate factor (1'000'000 * 2^20)/(1'000'000 + DRIFT_CONSTRAINT)
77  //       in order to replace two long long-divisions by right-shifts
78  #define MAX_DRIFT_FAC (long int) (1000000.0 / (1000000 - 2 * DRIFT_ADJUST *
        DRIFT_CONSTRAINT) * 1048576)
79  /**
80   * Precalculated factor for calculating current bounds
81   */
82  #define MIN_DRIFT_FAC (long int) (1000000.0 / (1000000 + 2 * (1 - DRIFT_ADJUST) *
        DRIFT_CONSTRAINT) * 1048576)
83
84  /**
85   * Precalculated factor for calculating bounds backwards (delta h < 0)
86   */
87  #define MAX_DRIFT_FAC_BW (long int) (1000000.0 / (1000000 - 2 * (1 - DRIFT_ADJUST) *
        DRIFT_CONSTRAINT) * 1048576)
88
89  /**
90   * Precalculated factor for calculating bounds backwards (delta h < 0)
91   */
92  #define MIN_DRIFT_FAC_BW (long int) (1000000.0 / (1000000 + 2 * DRIFT_ADJUST *
        DRIFT_CONSTRAINT) * 1048576)
93
94  /**
95   * Measured minimum round trip time.
96   */
97  #define SYNC_MIN_RTT 6
98
99  /**
100  * Maximum round trip time. Packets arriving with a higher rtt are to be discarded.
101  */
102 #define SYNC_MAX_RTT 115
103
104 /**
105  * Length of synchronisation bound in bytes
106  */
107 #define SYNC_BOUND_LEN 6
108
109  /*****************************************************************************
110  * ALGORITHM-SPECIFIC PREPROCESSOR CONSTANTS
111  *****************************************************************************/
112
113 /**
114  * Shortcut for the variable actually containg the currently relevant bounds.
115  */
116 #define SYNC_CURRENT_BOUNDS sync_globals.nearest_neighbour->bounds
117
118 /**
```

```
119   * Maximum number of entries stored in the linked list representing the neighbour
          BTnodes. (BP-ISA)
120   */
121  #define SYNC_MAX_NEIGHBOURS 10
122
123  #define SYNC_SYNC_DELAY 45
124
125  #define SYNC_SET_DELAY 57
126
127  #define SYNC_DELAY_UNCERTAINTY 15
128
129  /**
130   * Maximum computational delay in milliseconds
131   */
132  #define SYNC_COMPUTATIONAL_DELAY 1
133
134  /**
135   * Length of the BP_ISA timestamp in bytes. Contains 2 full lower and 2 full upper
          bounds.
136   * The length of BT_MAX_ACL_COM_PAYLOAD is reduced by this value.
137   */
138  #ifdef SYNC_RTT
139          #define SYNC_TSTAMP_LEN 28
140  #else
141          #define SYNC_TSTAMP_LEN 24
142  #endif
143
144  /**
145   * Marks the beginning of the second bound within a timestamp
146   */
147  #define SYNC_TSTAMP_OFFSET_1 6
148
149  /**
150   * Marks the beginning of the third bound within a timestamp
151   */
152  #define SYNC_TSTAMP_OFFSET_2 12
153
154  /**
155   * Marks the beginning of the fourth bound within a timestamp
156   */
157  #define SYNC_TSTAMP_OFFSET_3 18
158
159  /**
160   * Marks the beginning of the RTT-timestamp within a timestamp
161   */
162  #define SYNC_TSTAMP_OFFSET_RTT 24
163
164   /*****************************************************************************
165   * TYPES
166   *****************************************************************************/
167
168  /**
169   * @brief Represents a pair of bounds
170  */
171  typedef struct {
172          u_longlong lower;                       /**< Lower bound */
173          u_longlong upper;                       /**< Upper bound */
174          u_long last_update;                     /**< Local time when bounds were updated
                last */
175          char validity;                          /**< Validity of bounds (0: invalid, 1:
               valid) */
176  } syncbounds;
177
178  /**
179   * Return type of gettimeofday()
180   *
181   */
182  typedef struct _tv timeval;                     //should be defined in <time.h> rather
         than here
183
184  /**
185   * @struct _tv
186   * @brief Represents unix time in seconds and microseconds
187  */
188  struct _tv {
189          long tv_sec;            // seconds
190          long tv_usec;           // microseconds
191  };
192
193  /**
194   * Represents a neighbour BTnode
195   */
196  typedef struct _sync_nb sync_neighbour;
```

```
197
198 /**
199  * @struct _sync_nb
200  * @brief Represents a neighbour BTnode
201  * Make use of the typedef'd type sync_neighbour.\n
202  * A linked list is formed out of these elements in order to store
203  * the bounds of the last communication with the corresponding BTnode.
204  * The list is always ordered. The BTnode of the latest communication
205  * is at the beginning of the list, the BTnode of the oldest communication
206  * is at the end. Maximal SYNC_MAX_NEIGHBOURS neighbours are allowed. If
207  * already SYNC_MAX_NEIGHBOURS are stored in the list, the oldest (last)
208  * entry will be removed.
209 */
210 struct _sync_nb {
211         bt_addr_t address;
212         syncbounds bounds;
213         sync_neighbour * next;
214         #ifdef SYNC_RTT
215                 u_long last_rcvd;
216                 u_long last_send;
217                 u_char uncertainty;
218         #endif
219 };
220
221  /*****************************************************************************
222  * GLOBAL VARIABLES
223  *****************************************************************************/
224
225 struct _sync_globals_struct {
226
227         /**
228          * Timestamp to be sent
229          */
230         u_char sync_timestamp_to_send[SYNC_TSTAMP_LEN + 1];
231
232         /**
233          * The received timestamp
234          */
235         u_char sync_timestamp_rcvd[SYNC_TSTAMP_LEN + 1];
236
237         /**
238          * Pointer to the head of the neighbour list
239          */
240         sync_neighbour * nearest_neighbour;
241         /**
242          * Counts current neighbours
243          */
244         u_char neighbour_count;
245         /**
246          * The bluetooth address of the BTnode to which a packet is sent
247          */
248         bt_addr_t neighbour_to_send_addr;
249
250         /**
251          * Stores the LED pattern
252          */
253         char bitpattern;
254
255 } sync_globals;
256
257 /**
258  * Needed to synchronize the heartbeat
259  */
260 u_long btn_led_synch_offset;
261
262  /*****************************************************************************
263  * FUNCTION PROTOTYPES
264  *****************************************************************************/
265
266
267 /*****************************************************************************
268  *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
269  * HIGH LEVEL FUNCTIONS (APPLICATION INTERFACE)
270  *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
271  *****************************************************************************/
272
273 /*****************************************************************************
274  * rt_is_valid
275  *****************************************************************************/
276 /**
277  * @brief Returns validity of local bounds (High-level function)
278  */
279 char rt_is_valid(void);
```

```
280
281
282  /*****************************************************************************
283   * rt_get_bounds
284   *****************************************************************************/
285  /**
286   * @brief Returns current bounds (High-level function)
287   */
288  syncbounds rt_get_bounds(void);
289
290  /*****************************************************************************
291   * rt_get_average
292   *****************************************************************************/
293  /**
294   * @brief Gives the average of the local bounds (High-level function)
295   *
296   * Calculates the average of the current local bounds, if validity of bounds is 1.
297   * If validity of bounds is 0, the lower bound is passed back.
298   *
299   * @param av Pointer to the variable where result shall be stored
300   */
301  void rt_get_average(u_longlong * av);
302
303  /*****************************************************************************
304   * rt_get_monotonic
305   *****************************************************************************/
306  /**
307   * @brief Gives a strictly monotonically increasing clock value. (High-level function)
308   *
309   * Returns the current lower bound. This value lays within the guarantee interval
310   * and is strictly monotonically increasing, though the average error compared
311   * to the reference time is maximal.
312   *
313   * @param millis Pointer to the variable where result shall be stored
314   */
315  void rt_get_monotonic(u_longlong * millis);
316
317   /*****************************************************************************
318   * gettimeofday
319   *****************************************************************************/
320  /**
321   * @brief Get current unix time (High-level function)
322   *
323   * This is the BTnode implementation of the equally named C-function which
324   * gives the current unix time. The time is calculated as the average of the
325   * lower and the upper bound. Note that the granularity is in milliseconds, so
326   * the last three digits of the microseconds will always be zero. This is because
327   * the synchronization is not as accurate as to give microseconds, but the original
328   * format was kept for compatibility reasons.
329   *
330   * The function will return 1 if the time
331   * is valid, 0 if it is invalid, i.e. if the BTnode has valid respective invalide bounds
      .
332   * Communicating with BTnodes having valid bounds will turn the local bounds valid, as
      there is
333   * piggy-back synchronization.
334   *
335   * On an anchornode, this function will request for the timestamp from the connected PC.
336   *
337   * @param *tv Pointer to variable to store the time in.
338   * @param *tz Timezone. Not implemented, pass NULL.
339   *
340   * @return 1 if time is valid, 0 if time is invalid.
341   *
342   */
343  int gettimeofday(timeval * tv, void * tz);
344
345  /*****************************************************************************
346   * rt_get_millis
347   *****************************************************************************/
348  /**
349   * @brief Returns current unix milliseconds calculated by averaging the current bounds
350   */
351  u_long rt_get_millis(void);
352
353  /*****************************************************************************
354   * rt_get_seconds
355   *****************************************************************************/
356  /**
357   * @brief Returns current unix seconds calculated by averaging the current bounds
358   */
359  u_long rt_get_seconds(void);
360
```

```
361 /*****************************************************************************
362  *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
363  * SYNCHRONIZATION FUNCTIONS (INTERNAL)
364  *  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
365 *****************************************************************************/
366
367  /*****************************************************************************
368  * get_updated_bounds
369  *****************************************************************************/
370 /**
371  * @brief Same as update_bounds() except that the passed bounds remain unchanged
372  *
373  * The values of the bounds at reference time are calculated from the passed bounds,
374  * but are stored in a new syncbounds variable in order to leave the passed bounds
375  * unchanged.
376  *
377  * @param bounds The bounds to be updated
378  * @param reference_time The reference time the bounds should (have been) valid
379  * @return Bounds at reference time
380  */
381 syncbounds get_updated_bounds(syncbounds * bounds, u_long reference_time);
382
383
384  /*****************************************************************************
385  * rt_set_clock (with list initialization)
386  *****************************************************************************/
387 /**
388  * @brief Stores bounds into local bounds
389  *
390  * This function is called when the BTnode receives bounds for the first time.
391  * The own bluetooth address is removed from the linked list and the BTnode which
392  * sent the timestamp is inserted as nearest neighbour.
393  * The received bounds are stored and their validity is set to 1.\n
394  *
395  * On an anchornode, this function has no effect as an anchornode doesn't store
396  * any bounds.
397  */
398 void rt_set_clock(syncbounds * current_bounds_rcvd, sync_neighbour * nghb, u_long
      arrival_time, u_char uncertainty);
399
400 /*****************************************************************************
401  * sync_sync_bounds
402  *****************************************************************************/
403
404 /**
405  * @brief Performs the synchronization
406  *
407  * @param current_bounds_rcvd The received current bounds.
408  * @param previous_bounds_rcvd The past bounds.
409  * @param Bluetooth address of BTnodes from which bounds were received.
410  */
411 void sync_sync_bounds(syncbounds * current_bounds_rcvd, syncbounds *
      previous_bounds_rcvd, sync_neighbour * nghb, u_long arrival_time, u_char uncertainty
      );
412
413
414 /*****************************************************************************
415 * BP-ISA SPECIFIC FUNCTIONS
416 *****************************************************************************/
417
418 /*****************************************************************************
419 * rt_init (BP-ISA)
420 *****************************************************************************//**
421 * @brief Initializes the synchronization functions
422 *
423 * The linked list for BP-ISA is prepared. On anchornodes, the validity of the
424 * bounds is set to 1. The own bluetooth address is added to the linked list in
425 * order to ... TODO
426 */
427 void rt_init(struct btstack * stack);
428
429 /*****************************************************************************
430 * sync_assemble_timestamp (BP-ISA)
431 *****************************************************************************/
432 /**
433 * @brief Converts current and past bounds into a timestamp stored in
      sync_timestamp_to_send
434 */
435 void sync_assemble_timestamp(syncbounds * bounds_c, syncbounds * bounds_p);
436
437 /*****************************************************************************
438 * sync_dissassemble_timestamp (BP-ISA)
439 *****************************************************************************/
```

```
440  /**
441   * @brief Extracts the current and past bounds from the sync_timestamp_rcvd timestamp
442   */
443  void sync_dissassemble_timestamp(syncbounds * bounds_c, syncbounds * bounds_p);
444
445
446
447  /*****************************************************************************
448   * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
449   * COMM-EVENT LAYER FUNCTIONS (BT-HCI FUNCTIONS INTERFACE)
450   * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
451   *****************************************************************************/
452
453   /*****************************************************************************
454   * sync_generate_timestamp
455   *****************************************************************************/
456
457  /**
458   * @brief Called before sending packet
459   *
460   * This function is called just before a HCI_ACL_DATA_PACKET is sent to the bluetooth
461   * module in the function _bt_hci_send_pkt() in bt_hci_transport_uart.h.
462   * It updates the sync_timestamp_to_send in order to transmit the current
463   * timestamp by calling sync_generate_timestamp() with updated bounds.
464   *
465   * The bluetooth address of the destination device is needed in order to
466   * retrieve the right stored past bounds. If no such bounds exist (communicating for
467   * the first time with this BTnode), zero bounds with validity 0 are sent as past
468   * bounds.
469   *
470   * On an anchornode sync_generate_timestamp() is not called as the timestamp is
          requested
471   * from the connected PC and stored in sync_timestamp_to_send directly.
472   *
473   * @param bt_addr addr The bluetooth address to which the packet is sent to.
474   */
475  void sync_generate_timestamp(bt_addr_t addr);
476
477
478   /*****************************************************************************
479   * sync_process_timestamp
480   *****************************************************************************/
481
482  /**
483   * @brief Called after receiving packet
484   *
485   * This function is called just after the timestamp of a HCI_ACL_DATA_PACKET
486   * is read from the bluetooth module into sync_rcvd_timestamp in the function
487   * _bt_hci_get_pkt() in bt_hci_transport_uart.h.
488   *
489   * It calls sync_dissassemble_timestamp() to extract the bounds and intersects
490   * them with to localy stored bounds by calling sync_sync_bounds();
491   */
492  void sync_process_timestamp(bt_addr_t addr);
493
494
495  /*****************************************************************************
496   * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
497   * NEIGHBOUR LIST FUNCTIONS
498   * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
499  *****************************************************************************/
500
501   /*****************************************************************************
502   * print_neighbour_list
503   *****************************************************************************/
504  /**
505   * @brief Prints out the linked list of neighbouring BTnodes.
506   */
507  void print_neighbour_list(void);
508
509  /*****************************************************************************
510   * get_neighbour_by_bt_addr
511   *****************************************************************************/
512  /**
513   * @brief Searches the linked list for a specific neighbour.
514   *
515   * @param addr The bluetooth address of the neighbour which is searched for.
516   * @param nghb_ptr Pointer to the beginning of the linked list.
517   * @param neighbour_pre Pass a pointer to a pointer to a sync_neighbour in which the
          linked list element preceeding the one searched for shall be returned.
518   * @return Pointer to entry in the linked list.
519   */
```

```
520 sync_neighbour * get_neighbour_by_bt_addr(bt_addr_t addr, sync_neighbour * nghb_ptr,
        sync_neighbour ** neighbour_pre);
521
522  /*****************************************************************************
523         * create_new_neighbour
524         *****************************************************************************/
525  /**
526         * @brief Creates a new neighbour and returns a pointer to it.
527         */
528 sync_neighbour * create_new_neighbour(bt_addr_t addr);
529
530  /*****************************************************************************
531         * insert_neighbour
532         *****************************************************************************/
533  /**
534         * @brief Insertes a new neighbour.
535         */
536 void insert_neighbour(sync_neighbour * nghb_ptr);
537
538  /*****************************************************************************
539         * remove_neighbour
540         *****************************************************************************/
541  /**
542         * @brief Removes a neighbour temporarily from list without freeing its memory.
543         */
544 void remove_neighbour(sync_neighbour * nghb_pre_ptr);
545
546 /*****************************************************************************
547  * delete_last_neighbour (BP-ISA)
548  *****************************************************************************/
549 /**
550  * @brief Removes the last neighbour of the linked list and frees its memory.
551  */
552 int delete_last_neighbour(void);
553
554
555 /*****************************************************************************
556  * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
557  * AUXILIARY FUNCTIONS (INTERNAL)
558  * ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
559  *****************************************************************************/
560
561 /*****************************************************************************
562  * _sync_calculate_drift
563  *****************************************************************************/
564 /**
565  * @brief Internal function that performs a division by 2^20.
566  *
567  * This function is called by get_updated_bounds an computes current bounds
568  * from the bounds stored according to the formula given by the drift-constraint
569  * model.
570  *
571  * As the ATMega's instruction set doesn't offer a division, this done by an
572  * arithmetic right-shift. In order to get correctly rounded results, we first
573  * add 2^20 to the operand, because this bit is going to be the MSB of the bits
574  * neglected after the shift operation.
575  */
576 void _sync_calculate_drift(long long * delta_h, u_longlong * bound, long int drift_fac);
577
578
579 /*****************************************************************************
580  * update_neighbours (BP-ISA)
581  *****************************************************************************/
582 /**
583  * @brief Walks through the linked list and updates all the bounds.
584  *
585  * For all elements of the linked list,
586  * the values of the passed bounds at the time when the stored bounds were generated
587  * are calculated and intersected with the stored bounds. The resulting bounds are
        stored.
588  *
589  * @param bounds The bounds wich shall be intersected with the stored ones.
590  * @param nghb_ptr Pointer to the element where the updating shall be started.
591  */
592 void update_neighbours(syncbounds * bounds, sync_neighbour * nghb_ptr);
593
594 /*****************************************************************************
595  * sync_intersect_bounds (BP-ISA)
596  *****************************************************************************/
597 /**
598  * @brief Stored and candidate bounds are intersected and stored again.
599  *
600  * @param stored_bounds The bounds stored in the respective list entry. the
```

```
601   * intersected bounds are written back to these bounds.
602   * @param candidate_bounds The candidate bounds to intersect with. These bounds
603   * remain unmodified.
604   */
605  void _sync_intersect_bounds(syncbounds * stored_bounds, syncbounds * candidate_bounds);
606
607   /*****************************************************************************
608   * _lltobs
609   *****************************************************************************/
610  /**
611   * @brief Internal function which converts a long long to a byte string
612   * This function is used to convert a bound into characters for transmission.
613   *
614   */
615  int _lltobs(u_longlong * n, u_char * ps);
616
617   /*****************************************************************************
618   * _bstoll
619   *****************************************************************************/
620  /**
621   * @brief Internal function which converts a byte string to a long long
622   *
623   * This function is used to convert characters from transmission into a bound.
624   */
625  void _bstoll(u_char * ps, u_longlong * n);
626
627  /*****************************************************************************
628   * _print_dec
629   *****************************************************************************/
630  /**
631   * @brief Prints out a long long as decimal number
632   *
633   * This function can be used for debugging as printf() doesn't support
634   * the long long integer type.
635   */
636  void _print_dec(long long n);
637
638
639  void _sync_push_rtt(u_long n);
640
641  void _sync_pop_rtt(u_long *n);
642
643
644  #endif // SYNC_H_INCLUDED
```

A.2 sync.c

```
1   #include <sync/sync.h>
2
3   #ifdef SYNC
4   #include <bt/bt_acl_com.h> // bt_acl_com_print_bt_addr()
5
6   /***************************************
7    * list functions                      *
8    ***************************************/
9
10  void print_neighbour_list(void){
11          sync_neighbour * nghb_ptr = sync_globals.nearest_neighbour;
12          int count = 0;
13          while(nghb_ptr != NULL && count < SYNC_MAX_NEIGHBOURS + 1){
14                  printf_P(PSTR("\n%%_bt-address_of_element_%d:_"), count++);
15                  bt_acl_com_print_bt_addr(nghb_ptr->address);
16
17                          printf_P(PSTR("\n%%_validity:_%d"), nghb_ptr->bounds.validity);
18
19                  nghb_ptr = nghb_ptr->next;
20          }
21  }
22
23  sync_neighbour * get_neighbour_by_bt_addr(bt_addr_t addr, sync_neighbour * nghb_ptr,
        sync_neighbour ** neighbour_pre){
24          *neighbour_pre = NULL;
25          if(nghb_ptr == NULL){
26                  return NULL;
27          }
28          if(BD_ADDR_CMP(nghb_ptr->address, addr)){
29                  return nghb_ptr;
30          }
31          while(nghb_ptr->next != NULL){
32                  if(BD_ADDR_CMP(nghb_ptr->next->address, addr)){
```

```
33                          *neighbour_pre = nghb_ptr;
34                          return nghb_ptr->next;
35                  }
36                  NutEnterCritical();
37                  nghb_ptr = nghb_ptr->next;
38                  NutExitCritical();
39          }
40          return NULL;
41  }
42
43  sync_neighbour * create_new_neighbour(bt_addr_t addr){
44          sync_neighbour * new_element;
45          if(sync_globals.neighbour_count >= SYNC_MAX_NEIGHBOURS || (new_element =
                NutHeapAllocClear(sizeof(sync_neighbour))) == NULL){
46                  return NULL;
47          }else{
48                  BD_ADDR_CPY(new_element->address, addr);
49                  new_element->next = NULL;
50                  return new_element;
51          }
52  }
53
54  void insert_neighbour(sync_neighbour * nghb_ptr){
55          NutEnterCritical();
56          nghb_ptr->next = sync_globals.nearest_neighbour;
57          sync_globals.nearest_neighbour = nghb_ptr;
58          sync_globals.neighbour_count += 1;
59          NutExitCritical();
60  }
61
62  void remove_neighbour(sync_neighbour * nghb_pre_ptr){
63          NutEnterCritical();
64          nghb_pre_ptr->next = nghb_pre_ptr->next->next;
65          sync_globals.neighbour_count -= 1;
66          NutExitCritical();
67  }
68
69  int delete_last_neighbour(void){
70          sync_neighbour * tmp_ptr;
71          if(sync_globals.neighbour_count == 1){
72                  if(NutHeapFree(sync_globals.nearest_neighbour) == 0){
73                          NutEnterCritical();
74                          sync_globals.nearest_neighbour = NULL;
75                          sync_globals.neighbour_count = 0;
76                          NutExitCritical();
77                          return 0;
78                  }
79          }else if(sync_globals.neighbour_count > 1){
80                  NutEnterCritical();
81                  tmp_ptr = sync_globals.nearest_neighbour;
82                  while(tmp_ptr->next->next != NULL){
83                          tmp_ptr = tmp_ptr->next;
84                  }
85                  NutExitCritical();
86                  if(NutHeapFree(tmp_ptr->next) == 0){
87                          NutEnterCritical();
88                          tmp_ptr->next = NULL;
89                          sync_globals.neighbour_count -= 1;
90                          NutExitCritical();
91                          return 0;
92                  }
93          }
94          return -1;
95  }
96
97  void update_neighbours(syncbounds * bounds, sync_neighbour * nghb_ptr){
98          syncbounds tmp_bounds;
99          while(nghb_ptr != NULL){
100
101                         if(nghb_ptr->bounds.validity == 1){
102
103                 tmp_bounds = get_updated_bounds(bounds, nghb_ptr->bounds.last_update);
104                 _sync_intersect_bounds(&nghb_ptr->bounds, &tmp_bounds);
105
106                         }
107
108                 NutEnterCritical();
109                 nghb_ptr = nghb_ptr->next;
110                 NutExitCritical();
111         }
112 }
113
114 /**************************************
```

```
115  * rt_init, rt_set_clock                    *
116  ****************************************/
117
118  //should be done at startup
119  //       (especially sync_globals.global_bounds.validity has to be zero in order to
120  //       avoid unnecessary (and false) computations. furthermore, we want...
121
122  void rt_init(struct btstack* stack){
123      sync_globals.neighbour_count = 0;
124      sync_globals.nearest_neighbour = NULL;
125      bt_addr_t my_bt_addr;
126      //insert ourself as a neighbour in order to get non-bp-isa-similar
127      //    behaviour while the rt clock is unset.
128      bt_hci_read_bt_addr(stack, BT_HCI_SYNC, my_bt_addr);
129      //TODO: what can we do, if creating ourselves failed?
130      insert_neighbour(create_new_neighbour(my_bt_addr));
131
132      sync_globals.bitpattern = 0x5;
133      btn_led_heartbeat(10, (char) 0x1, 1);
134  }
135  void rt_set_clock(syncbounds * current_bounds_rcvd, sync_neighbour * nghb, u_long
         arrival_time, u_char uncertainty){
136          //message delay uncertainty
137          current_bounds_rcvd->lower -= uncertainty;
138          current_bounds_rcvd->upper += uncertainty;
139          current_bounds_rcvd->last_update = arrival_time;
140          //assign bounds
141          nghb->bounds = *current_bounds_rcvd;
142      //we don't need the dummy entry (with our own bd-address) any more
143      delete_last_neighbour();
144      //rtt
145      nghb->last_rcvd = RT_NOW;
146      //we must check if nghb isn't yet part of the list (this would result
147      //  in a cyclic list and unreachable storage...). this is only the
148      //  case, if the neighbour identified in sync_process_timestamp
149      //  was the nearest neighbour.
150      if(nghb != sync_globals.nearest_neighbour){
151              insert_neighbour(nghb);
152      }
153      //change heartbeat after setting rt clock
154      btn_led_synch_offset = RT_COARSE_AVERAGE(SYNC_CURRENT_BOUNDS.lower,
             SYNC_CURRENT_BOUNDS.upper) - MILLIS;
155      btn_led_heartbeat(31, (char) sync_globals.bitpattern, 1);
156  }
157
158  /****************************************
159   * get_updated_bounds                      *
160   ****************************************/
161
162  syncbounds get_updated_bounds(syncbounds * bounds, u_long reference_time){
163      //bounds to be returned; initialized with the current bounds
164      syncbounds retval_bounds = *bounds;
165      long long delta_h;
166      //time elapsed since the last update of the bounds
167      delta_h = reference_time - bounds->last_update;
168      //calculate new bounds
169      if(delta_h < 0){
170              _sync_calculate_drift(&delta_h, &retval_bounds.lower, MAX_DRIFT_FAC_BW);
171              _sync_calculate_drift(&delta_h, &retval_bounds.upper, MIN_DRIFT_FAC_BW);
172      }else{
173              _sync_calculate_drift(&delta_h, &retval_bounds.lower, MIN_DRIFT_FAC);
174              _sync_calculate_drift(&delta_h, &retval_bounds.upper, MAX_DRIFT_FAC);
175      }
176      return retval_bounds;
177  }
178
179  //integer division by right-shift, preceded by adding 2^20 (this would have been
180  //       the msb of the discarded fractional part)
181  void _sync_calculate_drift(long long * delta_h, u_longlong * bound, long int drift_fac){
182          *bound += (long long) ((long long) (*delta_h * drift_fac) + 1048576) >> 20;
183  }
184
185
186  void _sync_intersect_bounds(syncbounds * stored_bounds, syncbounds * candidate_bounds){
187      //disjoint intervals: received bounds lay above or beneath our bounds:
188      //  => we'll just discard them.)
189      if(stored_bounds->lower >= candidate_bounds->upper || stored_bounds->upper <=
             candidate_bounds->lower){
190              return;
191      //if the intervals are overlapping: do the intersection
192      }else{
193              //lower_bound = max(lower_bound, lower_bound_received)
194              if(stored_bounds->lower < candidate_bounds->lower){
```

```
195                     stored_bounds->lower = candidate_bounds->lower;
196             }
197             //upper_bound = min(upper_bound, lupper_bound_received)
198                 if(stored_bounds->upper > candidate_bounds->upper){
199                         stored_bounds->upper = candidate_bounds->upper;
200             }
201     }
202 }
203
204 /*****************************************
205  * sync_sync_bounds                      *
206  *****************************************/
207 //does the synchronization of the bounds
208
209 void sync_sync_bounds(syncbounds * current_bounds_rcvd, syncbounds *
        previous_bounds_rcvd, sync_neighbour * nghb, u_long arrival_time, u_char uncertainty
        ){
210     //take into account uncertainty over message delay
211         current_bounds_rcvd->lower -= uncertainty;
212         current_bounds_rcvd->upper += uncertainty;
213     //only if previous bounds are valid (useless to update with invalid (infinite)
            bounds)
214     if(nghb->bounds.validity == 1){
215             if(previous_bounds_rcvd->validity == 1){
216                 //insert update time of 'previous_bounds_rcvd' in order to simplify
217                 //      update_neighbours (~'mutual communication event')
218                 previous_bounds_rcvd->last_update = nghb->bounds.last_update;
219                 //add uncertainty from the last encounter with nghb to the previous
                        bounds sent by it
220                 previous_bounds_rcvd->lower -= nghb->uncertainty;
221                 previous_bounds_rcvd->upper += nghb->uncertainty;
222                 //store the uncertainty from this communication with nghb
223                 nghb->uncertainty = uncertainty;
224                 //update neighbourhood
225                 if(nghb == sync_globals.nearest_neighbour){
226                         update_neighbours(previous_bounds_rcvd, sync_globals.
                            nearest_neighbour->next);
227                 }else{
228                         update_neighbours(previous_bounds_rcvd, sync_globals.
                            nearest_neighbour);
229                 }
230                 //the neighbour we're talking with doesn't need to be updated because
                        the time
231                 //      corresponding to its previous bounds should be the same on both
                        nodes communicating
232                 _sync_intersect_bounds(&nghb->bounds, previous_bounds_rcvd);
233             }
234         }
235     //get current bounds
236     nghb->bounds = get_updated_bounds(&sync_globals.nearest_neighbour->bounds,
            arrival_time);
237     nghb->bounds.last_update = arrival_time;
238     //determine new local bounds
239     _sync_intersect_bounds(&nghb->bounds, current_bounds_rcvd);
240
241     //if we have removed the neighbour:
242     if(nghb != sync_globals.nearest_neighbour){
243             //re-insert neighbour into neighbourhood as nearest neighbour
244             insert_neighbour(nghb);
245     }
246     //update all neighbours except the nearest with the new bounds
247         update_neighbours(&nghb->bounds, nghb->next);
248     //synchronized heartbeat
249     btn_led_synch_offset = RT_COARSE_AVERAGE(nghb->bounds.lower, nghb->bounds.upper) -
            MILLIS;
250 }
251
252 /*****************************************************
253  * high-level functions (application interface)  *
254  *****************************************************/
255
256 char rt_is_valid(void){
257         return SYNC_CURRENT_BOUNDS.validity;
258 }
259
260 syncbounds rt_get_bounds(void){
261         return get_updated_bounds(&SYNC_CURRENT_BOUNDS, RT_NOW +
                SYNC_COMPUTATIONAL_DELAY);
262 }
263
264 void rt_get_average(u_longlong * av){
265     syncbounds bounds = get_updated_bounds(&SYNC_CURRENT_BOUNDS, RT_NOW +
            SYNC_COMPUTATIONAL_DELAY);
```

```
266     if(bounds.validity == 0){
267             //rt clock hasn't been set yet: return the lower bound (this is a valid
268             //  lower guaranteed rt-bound and is monotonicaly increasing.
269             *av = bounds.lower;
270     }else{
271             //compute the average with a leftshift
272             *av = (bounds.lower + bounds.upper) >> 1;
273 }
274 }
275
276 void rt_get_monotonic(u_longlong * millis){
277     syncbounds bounds = get_updated_bounds(&SYNC_CURRENT_BOUNDS, RT_NOW +
            SYNC_COMPUTATIONAL_DELAY);
278     //the lower guaranteed rt-bound is monotonicaly increasing.
279     *millis = bounds.lower;
280 }
281
282 int gettimeofday(timeval * tv, void * tz){
283     //as tz is obsolete, we neglect it. just pass NULL for the second argument.
284     u_longlong millistamp;
285     rt_get_average(&millistamp);
286     tv->tv_sec = (long) (millistamp / 1000);
287     //granularity is only milliseconds
288     tv->tv_usec = (long) ((millistamp % 1000) * 1000);
289     return rt_is_valid() - 1;
290 }
291
292 u_long rt_get_millis(void){
293     u_longlong ms;
294     rt_get_average(&ms);
295     return (u_long) ms;
296 }
297
298 u_long rt_get_seconds(void){
299     u_longlong sec;
300     rt_get_average(&sec);
301     return (u_long) (sec / 1000);
302 }
303
304 /**************************************
305  * sync_generate_timestamp            *
306  **************************************/
307 // handles request from 'bt_hci_transport_uart'
308 void sync_generate_timestamp(bt_addr_t addr){
309     sync_neighbour * neighbour_to_send;
310         syncbounds send_bounds_p;
311         send_bounds_p.validity = 0;
312         u_long now = RT_NOW + SYNC_COMPUTATIONAL_DELAY;
313     //get current bounds
314     syncbounds send_bounds_c = get_updated_bounds(&SYNC_CURRENT_BOUNDS, now);
315     neighbour_to_send = get_neighbour_by_bt_addr(addr, sync_globals.nearest_neighbour,
            NULL);
316     //if neighbour already exists:
317     if(neighbour_to_send != NULL){
318             if(neighbour_to_send->bounds.validity == 1){
319                     //  send previous bounds
320                     send_bounds_p = neighbour_to_send->bounds;
321                     //update the neighbour's bounds (~'mutual communication event')
                            after sending
322                     neighbour_to_send->bounds = get_updated_bounds(&neighbour_to_send->
                            bounds, now);
323             }
324     //otherwise (i.e. if we're talking to an new, unknown neighbour):
325     }else{
326         //create new neighbour if possible
327     neighbour_to_send = create_new_neighbour(addr);
328     if(neighbour_to_send == NULL){
329         //      or free some memory space by removing a neighbour
330         delete_last_neighbour();
331         //      and try it again. (No further error handling is done)
332         neighbour_to_send = create_new_neighbour(addr);
333         if(neighbour_to_send == NULL) { ;
334                 btn_led_clear_pattern_queue();
335                 btn_led_add_pattern(BTN_LED_PATTERN_HALF, 0,1, 100);
336         }
337     }
338         //we put best-effort accuracy
339         neighbour_to_send->bounds = send_bounds_c;
340     //put the new neighbour on the head of the list (where else could we put it?)
341     insert_neighbour(neighbour_to_send);
342     //last_update means rather 'last comm event' than 'last recv event' now
343     neighbour_to_send->bounds.last_update = now;
344     //rtt
```

```
345     neighbour_to_send->last_send = RT_NOW;
346         if(neighbour_to_send->last_rcvd == 0) {
347                 _sync_push_rtt(0);
348         }else{
349                 _sync_push_rtt(neighbour_to_send->last_send - neighbour_to_send->
                        last_rcvd);
350         }
351     //do the conversions into chars
352     sync_assemble_timestamp(&send_bounds_c, &send_bounds_p);
353     //as we updated our current bounds, we'll update the led synchro global as well
354     btn_led_synch_offset = RT_COARSE_AVERAGE(send_bounds_c.lower, send_bounds_c.upper)
            - MILLIS;
355 }
356
357 /***************************************
358  * sync_process_timestamp              *
359  ***************************************/
360 // handles request from 'bt_hci_transport_uart'
361 void sync_process_timestamp(bt_addr_t addr){
362         sync_neighbour * nghb;
363         sync_neighbour * nghb_pre = NULL;
364         u_long rtt;
365         u_char offset;
366     syncbounds rcvd_bounds_c;
367     syncbounds rcvd_bounds_p;
368
369     sync_dissassemble_timestamp(&rcvd_bounds_c, &rcvd_bounds_p);
370     //get the neighbour we're communicating with from the linked list
371     if((nghb = get_neighbour_by_bt_addr(addr, sync_globals.nearest_neighbour, &nghb_pre)
            ) == NULL){
372         //if there is no entry for this neighbour:
373         //      try to allocate a new entry; otherwise
374         if((nghb = create_new_neighbour(addr)) == NULL){
375             //  delete the last entry and
376             delete_last_neighbour();
377             //  try it again
378             nghb = create_new_neighbour(addr);
379             if(nghb == NULL){
380                     btn_led_clear_pattern_queue();
381                     btn_led_add_pattern(BTN_LED_PATTERN_HALF, 0,1, 100);
382             }
383             }
384     //we don't insert the new neighbour yet...
385     }else{
386         //...we even remove the neighbour temporarily from list, if it exists
387         // and if it isn't the nearest (the head of the linked list)
388         if(nghb_pre != NULL){
389             remove_neighbour(nghb_pre);
390         }
391     }
392     nghb->last_rcvd = RT_NOW;
393     if(nghb->last_send == 0){
394         //we want bounded rtt from the first communication on. another
395         //       possibility is to assign rtt the value zero here and thus
396         //       accept every packet from a new neighbour. it's a tradeoff
397         //       between accuracy and performance.
398         rtt = SYNC_MAX_RTT;
399     }else{
400             _sync_pop_rtt(&rtt);
401             rtt = nghb->last_rcvd - nghb->last_send - rtt;
402     }
403     //iff the bounds received are valid and the rtt is small enough
404     if(rcvd_bounds_c.validity == 1 && rtt < SYNC_MAX_RTT){
405                 offset = (u_char) rtt;
406                 //the rtt cannot be less than two times the minimum rtt actually.
407                 //      even though, we'll handle this exception safely
408                 if(offset < (u_char) 2*SYNC_MIN_RTT){
409                 offset = (u_char) SYNC_MIN_RTT;
410                 }else{
411                 //offset = (rtt / 2)
412                 asm volatile(
413                         "lsr_%0" "\n\t"
414                         "adc_%0,___zero_reg__" "\n\t" :
415                         "=r" (offset) :
416                         "0" (offset)
417                 );
418                 }
419         //if this is the first communication with a neighbour knowing valid bounds:
420         if(SYNC_CURRENT_BOUNDS.validity == 0){
421             //set the rt clock and return (rt_set_clock handles nghb's insertion)
422             //return rt_set_clock(&rcvd_bounds_c, nghb, RT_NOW - SYNC_MIN_RTT, ((u_char)
                    rtt - (u_char) (2 * SYNC_MIN_RTT)));
```

```
423             return rt_set_clock(&rcvd_bounds_c, nghb, RT_NOW - offset, offset - (u_char)
                    SYNC_MIN_RTT);
424         }else{
425                 //do synchronization
426                     //sync_sync_bounds(&rcvd_bounds_c, &rcvd_bounds_p, nghb, RT_NOW -
                            SYNC_MIN_RTT, ((u_char) rtt - (u_char) (2 * SYNC_MIN_RTT)));
427                     sync_sync_bounds(&rcvd_bounds_c, &rcvd_bounds_p, nghb, RT_NOW -
                            offset, offset - (u_char) SYNC_MIN_RTT);
428                 }
429     }else{
430             //for SYNC_RTT (we'd like to build the list even if uninitialized
431             //  or if bounds are invalid) we have to insert nghb now:
432         if(nghb != sync_globals.nearest_neighbour){
433                 if(SYNC_CURRENT_BOUNDS.validity == 1){
434                 //once we've got valid bounds, we insert neighbours always with
435                 //      valid (best-effort) bounds as well
436                 nghb->bounds = get_updated_bounds(&SYNC_CURRENT_BOUNDS, nghb->last_rcvd)
                        ;
437                 nghb->bounds.last_update = nghb->last_rcvd;
438             }
439             insert_neighbour(nghb);
440         }
441
442 /**************************************
443  * auxiliary functions                *
444  **************************************/
445
446 void sync_assemble_timestamp(syncbounds * bounds_c, syncbounds * bounds_p) {
447     _lltobs(&bounds_c->lower, sync_globals.sync_timestamp_to_send);
448     _lltobs(&bounds_c->upper, sync_globals.sync_timestamp_to_send + SYNC_TSTAMP_OFFSET_1
            );
449     _lltobs(&bounds_p->lower, sync_globals.sync_timestamp_to_send + SYNC_TSTAMP_OFFSET_2
            );
450     _lltobs(&bounds_p->upper, sync_globals.sync_timestamp_to_send + SYNC_TSTAMP_OFFSET_3
            );
451
452     sync_globals.sync_timestamp_to_send[SYNC_TSTAMP_OFFSET_1 - 1] |= (u_char) (bounds_c
            ->validity << 4);
453     sync_globals.sync_timestamp_to_send[SYNC_TSTAMP_OFFSET_3 - 1] |= (u_char) (bounds_p
            ->validity << 4);
454 }
455
456 void sync_dissassemble_timestamp(syncbounds * bounds_c, syncbounds * bounds_p) {
457     if((bounds_p->validity = (sync_globals.sync_timestamp_rcvd[SYNC_TSTAMP_OFFSET_3
            - 1]) >> 4) == 1){
458         _bstoll(sync_globals.sync_timestamp_rcvd + SYNC_TSTAMP_OFFSET_2, &bounds_p->
                lower);
459         _bstoll(sync_globals.sync_timestamp_rcvd + SYNC_TSTAMP_OFFSET_3, &bounds_p->
                upper);
460         }
461     if((bounds_c->validity = (sync_globals.sync_timestamp_rcvd[SYNC_TSTAMP_OFFSET_1
            - 1]) >> 4) == 1){
462             _bstoll(sync_globals.sync_timestamp_rcvd, &bounds_c->lower);
463             _bstoll(sync_globals.sync_timestamp_rcvd + SYNC_TSTAMP_OFFSET_1, &bounds_c->
                    upper);
464     }
465 }
466
467 // converts u_longlong to u_char[SYNC_BOUND_LEN]
468 // returns -1, if not possible
469 // ATTENTION: n will become unusable, if SYNC_BOUND_LEN != 6
470 int _lltobs(u_longlong * n, u_char * ps) {
471         u_char * ptr = (u_char *) n;
472         ps[0] = *(ptr + 0);
473         ps[1] = *(ptr + 1);
474         ps[2] = *(ptr + 2);
475         ps[3] = *(ptr + 3);
476         ps[4] = *(ptr + 4);
477         ps[5] = *(ptr + 5);
478
479         return 0;
480 }
481
482 void _sync_push_rtt(u_long n){
483     u_char * ptr = (u_char *) &n;
484     sync_globals.sync_timestamp_to_send[SYNC_TSTAMP_OFFSET_RTT] = *(ptr + 0);
485     sync_globals.sync_timestamp_to_send[SYNC_TSTAMP_OFFSET_RTT + 1] = *(ptr + 1);
486     sync_globals.sync_timestamp_to_send[SYNC_TSTAMP_OFFSET_RTT + 2] = *(ptr + 2);
487     sync_globals.sync_timestamp_to_send[SYNC_TSTAMP_OFFSET_RTT + 3] = *(ptr + 3);
488 }
489
490 // converts byte array to u_longlong
491 void _bstoll(u_char * ps, u_longlong * n){
```

```
492     ps[SYNC_BOUND_LEN - 1] &= (u_char) 0x03;
493
494     u_char * ptr = (u_char *) n;
495     *(ptr + 0) = ps[0];
496     *(ptr + 1) = ps[1];
497     *(ptr + 2) = ps[2];
498     *(ptr + 3) = ps[3];
499     *(ptr + 4) = ps[4];
500     *(ptr + 5) = ps[5];
501     *(ptr + 6) = (u_char) 0;
502     *(ptr + 7) = (u_char) 0;
503 }
504
505 void _sync_pop_rtt(u_long *n){
506     u_char * ptr = (u_char *) n;
507
508     *(ptr + 0) = sync_globals.sync_timestamp_rcvd[SYNC_TSTAMP_OFFSET_RTT];
509     *(ptr + 1) = sync_globals.sync_timestamp_rcvd[SYNC_TSTAMP_OFFSET_RTT + 1];
510     *(ptr + 2) = sync_globals.sync_timestamp_rcvd[SYNC_TSTAMP_OFFSET_RTT + 2];
511     *(ptr + 3) = sync_globals.sync_timestamp_rcvd[SYNC_TSTAMP_OFFSET_RTT + 3];
512 }
513
514 // prints unsigned long long as decimal, as printf somehow doesn't work
515 void _print_dec(long long n) {
516     /* print in decimal */
517     char chars[32];
518     int count = 0;
519     if(n < 0){
520         printf_P(PSTR("-"));
521     }
522     do{
523         chars[count] = abs(n % 10);
524         n = n / 10;
525         count = count + 1;
526         }while(n);
527     count = count - 1; /* just incremented above */
528
529     while(count >= 0){
530         printf_P(PSTR("%d"), chars[count]);
531         count = count - 1;
532     }
533 }
534
535
536 #endif // SYNC
```