CARL VON OSSIETZKY universität OLDENBURG

DEPARTMENT FÜR INFORMATIK
SYSTEMSOFTWARE UND VERTEILTE SYSTEME

Quantifizierung des Selbststabilisierungsverhaltens eines selbststabilisierenden Sensor-/Aktorennetzwerkes

Bachelorarbeit

30. September 2013

Marius Hacker
Klingenbergstraße 50A
26133 Oldenburg

Erstprüfer Prof. Dr.-Ing. Oliver Theel
Zweitprüfer Dipl.-Inform. Eike Möhlmann

Erklärung zur Urheberschaft

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe.

Oldenburg, den 30. September 2013

Marius Hacker

Mit Sensornetzwerken lassen sich autonom Sensordaten in Regionen messen. Bei autonomen Systemen ist die Behandlung von Fehlern wichtig. Zur Behandlung von Fehlern gibt es sogenannte Fehlertoleranzkonzepte. Eins dieser Fehlertoleranzkonzepte ist die Selsbststabilisierung. Ein selbststabilisierendes Sensornetzwerk ist in der Lage, sich nach Auftreten eines Fehlers selbst wieder in einen funktionsfähigen Zustand zu versetzen. Diese Arbeit befasst sich mit der Entwicklung und Implementierung eines solchen selbststabilisierenden Systems. Dieses System soll den selbststabilisierenden Mechanismus in einer nachvollziehbaren Weise für einen Betrachter darstellen. Zu diesem Zweck wurde ein Sensornetzwerk entworfen, welches den Mittelwert über alle Sensorknoten in diesem Netzwerk berechnet und auf einem Display ausgibt. Um diesen Algorithmus auf dem Sensornetzwerk zu implementieren, sind einige Anpassungen an diesem nötig. Aus diesem Grund wird das Selbststabilisierungsverhalten des Sensornetzwerkes innerhalb dieser Arbeit quantifiziert.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Für das Sammeln von Sensordaten in einer Region werden meist sogenannte Sensornetzwerke eingesetzt. Ein Sensornetzwerk besteht aus mehreren Sensorknoten. Diese batteriebetriebenen Kleinstrechner können per Funk miteinander kommunizieren und so gesammelte Informationen austauschen. Auf den Sensorknoten werden meist besonders energiesparende Betriebssysteme eingesetzt, sodass diese über Monate ohne Wechsel oder Laden der Energiequelle funktionieren. Sensorknoten können viele verschiedene Daten sammeln. Sie können beispielsweise mit Helligkeits-, Feuchtigkeits-, Temperatur-, oder Erschütterungssensoren ausgestattet werden. Ein Anwendungsszenario wäre die Messung von seismischen Aktivitäten in einer Region. Dieses Sensornetzwerk könnte als Warnsystem vor Erdbeben dienen, oder die Daten über einen längeren Zeitraum sammeln, damit diese dann später ausgewertet werden können. Ein solches Sensornetzwerk muss über Monate vollkommen autonom agieren können.

Ein wichtiger Teil bei der Entwickelung eines autonomen Systems ist die Behandlung von auftretenden Fehlern. Ein nicht behandelter Fehler könnte ansonsten die Funktionalität einschränken und das Eingreifen einer Person erfordern. Können schwer vorhersehbare Fehler auftreten, ist es sinnvoll ein Fehlertoleranzkonzept einzusetzen. Selbststabilisierung ist ein solches Fehlertoleranzkonzept.
Ein System welches selbststabilisierend ist, ist in der Lage sich ohne explizite Initialisierung in einen funktionsfähigen Zustand zu versetzen. Das System ist gegenüber sogenannten transienten Fehlern unempfindlich. Transiente Fehler, sind Fehler, die nur vorübergehend auftreten. Ein selbststabilisierendes System versetzt sich nach gewisser Zeit von selbst wieder in einen arbeitsfähigen Zustand.

Die Hauptaufgabe dieser Arbeit besteht in der Entwicklung und der Implementierung eines autonomen Sensornetzwerkes, das sich selbst im obigen Sinne stabilisiert. In Kapitel 3 wird auf die Grundlagen der Arbeit eingegangen, während in Kapitel 4 die Anforderungen eines solchen Systems genauer spezifiziert werden. Der Entwurf und die Implementierung des Systems wird in Kapitel 5 und 7 behandelt.

2 Zielsetzung der Arbeit

Bei der Implementierung eines selbststabilisierenden Systems müssen unter Umständen Kompromisse bezüglich der Modellannahme getroffen werden. Es soll untersucht werden in wie weit sich diese auf die Selbststabilisierung des Systems auswirken. Dazu wird der Unterschied zwischen erwartetem Ergebnis und experimentellem Laufzeitergebnis verglichen. Bei der Modellierung von selbststabilisierenden Systemen wird oft das Vorhandensein eines zentralen Steuerprogramms (engl. Scheduler) und die Atomarität komplexer Befehlsausführungsfolgen auf den Sensorknoten vorausgesetzt. Durch Untersuchung der Kompromisse bezüglich der Modellannahme soll quantitativ/qualitativ untermauert werden, welche Auswirkung diese bei einem konkreten System hat.
Das Sensornetzwerk soll die Selbststabilisierung in einer gut nachvollziehbaren Form darstellen. Denkbar wäre Beispielsweise eine Visualisierung mithilfe an den Sensorknoten angebrachter LCDs (Liquid-Crystal-Displays), sodass der arbeitende Mechanismus für außenstehende Beobachter gut nachvollziehbar ist.

3 Grundlagen

In diesem Kapitel werden die Grundlagen dieser Arbeit erläutert. Im ersten Teil wird der Begriff Selbststabilisierung genauer definiert und anhand eines Beispiels erklärt. Im zweiten Teil wird die eingesetzte Hardware genauer beschrieben.

3.1 Selbststabilisierung

Selbststabilisierung ist ein Fehlertoleranzkonzept und wurde 1974 von Edsger W. Dijkstra erfunden. In seinem Artikel „Self stabilizing systems in spite of distributed control“ [Dij74] beschreibt Dijkstra erstmals die Eigenschaften eines solchen selbststabilisierenden Systems.

3.1.1 Ein System

Ein System besteht aus einer oder mehreren Maschinen und Verbindungen zwischen diesen Maschinen. Die Maschinen werden im Folgenden als Knoten bezeichnet, wenn von Systemen in Form von Sensorknoten die Rede ist. Über die Verbindungen können die Prozesse untereinander kommunizieren. Die Knoten mit denen sie direkt kommunizieren können, werden als Nachbarn bezeichnet.

3.1.2 Definition Selbststabilisierung

Selbststabilisierung ist eine Form der Fehlertoleranz. Ist ein Algorithmus selbststabilisierend kann er sich nach Auftreten eines Fehlers selbstständig wieder in einen stabilen Zustand versetzen.
In dem Artikel [Sch93] wird Selbststabilisierung wie folgt definiert. Wir betrachten ein Prädikat $\mathcal{P}$. Es beschreibt den stabilen Zustand des Systems $\mathcal{S}$. Wir nennen $\mathcal{S}$ selbststabilisierend, wenn folgende zwei Eigenschaften erfüllt sind:

1. **Konvergenz:** Von einem beliebigen Zustand gestartet wird $\mathcal{P}$ in endlich vielen Schritten erreicht

2. **Geschlossenheit:** In einem Zustand gestartet, der $\mathcal{P}$ erfüllt, werden alle Folgezustände auch von $\mathcal{P}$ erfüllt

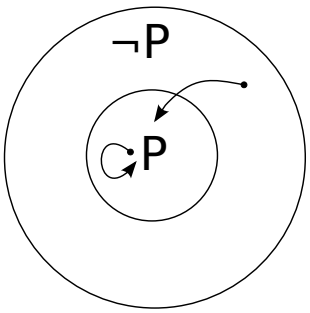


Abbildung 3.1: Selbststabilisierende Eigenschaften

Die beiden Eigenschaften sind in Abbildung 3.1 dargestellt. Die erste Eigenschaft stellt sicher, dass das System in einem gültigen Zustand bleibt, solange kein Fehler passiert. Tritt ein Fehler auf, so ist durch Eigenschaft zwei sichergestellt, dass das System wieder in einen gültigen Zustand versetzt wird. Da das System eine gewisse Zeit benötigt, um sich zu stabilisieren, kann es sein, dass es nicht in einen stabilen Zustand kommt, wenn Fehler permanent auftreten. Ein System kann nur gegen transiente Fehler selbststabilisierend sein, da ein System eine gewisse Zeit benötigt, um in einen sicheren Zustand zu gelangen. Tritt innerhalb dieser Zeit immer wieder ein Fehler auf, so kann sich das System nicht stabilisieren.

3.1.3 Beispiel Selbststabilisierung

Nachfolgend wird das Konzept der Selbststabilisierung anhand eines einfachen Beispiels genauer erläutert. Dazu betrachten wir ein System S. Auf S wird ein Algorithmus A ausgeführt, der eine ganzzahlige Variable v benutzt. v kann ganzzahlige Werte von 0 bis 255 annehmen. Der Algorithmus soll selbststabilisierend gegenüber einem Prädikat P sein. In diesem Beispiel ist P erfüllt, wenn die Variable v den Wert 5 hat. Der Algorithmus A ist in einem stabilen Zustand, wenn P erfüllt ist. Ist P nicht erfüllt, so ist A in keinem stabilen Zustand.
Der auf dem System ausgeführte Algorithmus ist in Abbildung 3.2 zu sehen. Der Wert der Variablen v wird nicht initialisiert, sodass v beim Starten des Algorithmus einen beliebigen Wert annimmt. Wie für selbststabilisierende Algorithmen üblich, wird der Algorithmus durchgängig ausgeführt, hier durch eine Schleife mit einer Bedingung, die immer erfüllt ist, realisiert. Der Algorithmus überprüft abwechselnd ob der Wert von v größer oder kleiner als 5 ist. Ist er größer, so wird die Variable um den Wert 1 verringert, ist er kleiner, so wird v um den Wert 1 erhöht. Ist der Wert der Variablen v gleich 5, verändert der Algorithmus die Variable nicht.

Anhand der im Abschnitt 3.1.2 vorgestellten Eigenschaften, wird nun überprüft

ob der Algorithmus selbststabilisierend ist. Die erste Eigenschaft (Konvergenz) besagt, dass von einem beliebigen Zustand gestartet, P in endlich vielen Schritten erfüllt wird. Die Zustände unterscheiden sich in diesem Fall durch den Wert von v. Zunächst wird der trivialen Fall $v = 5$ betrachtet. Der Algorithmus benötigt keinen Schritt um zu stabilisieren, die Eigenschaft ist erfüllt. Ist der Algorithmus in einem Zustand indem v einen größeren Wert als 5 hat, wird v solange um den Wert 1 verringert bis v den Wert 5 hat. Ist der Algorithmus in einem Zustand in dem v einen kleineren Wert als 5 hat, wird v erhöht, bis es den Wert 5 annimmt. Die Schritte s, die dazu nötig sind lassen sich mit der Formel $s = v - 5$ bestimmen und es sind somit endlich viele. Die zweite Eigenschaft (Geschlossenheit) ist erfüllt, da der Algorithmus den Wert von v nicht verändert, solange er 5 ist. Damit ist der Algorithmus A selbststabilisierend gegenüber dem Prädikat P.

3.2 Sensornetzwerk

Das für diese Arbeit benutzte Sensornetzwerk besteht aus mehreren Sensorknoten der Firma Maxford. Im Folgenden sollen diese Sensorknoten und die zusätzlich benutzte Hardware beschrieben werden.

3.2.1 Sensorknoten

Für diese Arbeit werden Sensorknoten vom Typ MTM-CM5000MSP [inc] der Firma Maxford verwendet (Abbildung 3.3). Die Sensorknoten sind baugleich mit den, in vielen Projekten genutzten, TelosB/Tmote Sky Sensorknoten. Als Mikrocontroller kommt ein Texas Instruments MSP430 zum Einsatz. Der MSP430 hat 48 KByte Programmspeicher, 10 KByte Arbeitsspeicher und 1 MByte externen Speicher. Der Sensorknoten besitzt einen CC2420 Funkchip für die drahtlose Kommunikation, Helligkeits-, Temperatur- und Feuchtigkeitssensoren und drei verschieden farbige LEDs. Außerdem befinden sich auf der Platine 16-Pins die unter anderem als Analog-Digital-Wandler verwendet werden können.

```
while TRUE do
    if v > 5 then
        v ← v − 1
    end if
    if v < 5 then
        v ← v + 1
    end if
end while
```

Abbildung 3.2: Beispielalgorithmus A

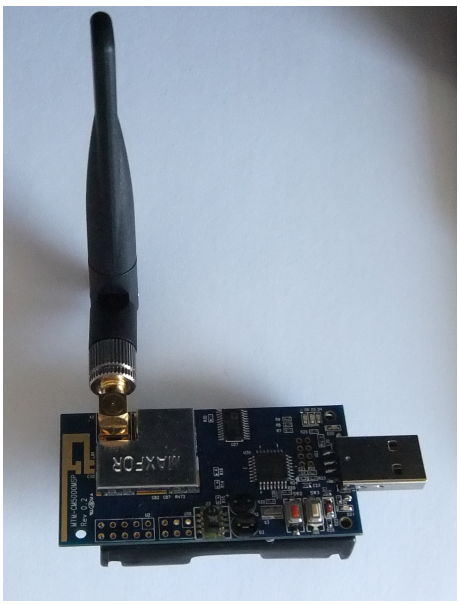


Abbildung 3.3: Sensorknoten

Die Sensorknoten haben einen USB-Anschluss über den die Knoten programmiert werden können. Die Stromversorgung kann direkt über diesen USB-Anschluss geschehen, oder mit zwei Batterien, die in eine Batteriehalterung unterhalb des Knotens eingelegt werden. Der Sensorknoten läuft mit einer Spannung von 3.3V.

Als Betriebssystem werden auf Sensorknoten oft TinyOS [Tin] oder Contiki-OS [Con] eingesetzt. Diese beiden Betriebssysteme wurden für den Telos-B/Tmote Sky Sensorknoten portiert. In dieser Arbeit wird Contiki-OS in der Version 2.6 verwendet. Contiki-OS ist ein ereignisgesteuertes, energieeffizientes Betriebssystem, welches speziell für Sensornetzwerke optimiert ist. Es beherrscht den IP-Network-Stack mit Protokollen wie UDP, TCP und HTTP. Außerdem beherrscht es den Rime-Network-Stack, ein ressourcensparendes energieeffizientes Netzwerkprotokoll, welches in dieser Arbeit verwendet wird. Contiki-OS ist in der Programmiersprache C geschrieben.

3.2.2 Display

An jedem Sensorknoten ist ein Display [expa] angeschlossen, dass Informationen über den Zustand des Knotens ausgibt. Das LCD kann je 16 Zeichen auf zwei Zeilen ausgeben. Die Zeichen werden in weiß auf blauem Hintergrund dargestellt. Zum Betrieb benötigt das Display eine Versorgungsspannung von 5 Volt. Mit den 3,3 Volt, die der Sensorknoten liefert, kann das Display nicht direkt betrieben werden. Zwischen den Sensorknoten und das Display wird ein Spannungswandler [expb] geschaltet, der in der Lage ist eine Spannung zwischen 2,7 Volt und 11,8 Volt auf 5 Volt zu wandeln. Das Display kann

mithilfe des Spannungswandlers den Sensorknoten als Stromquelle benutzen und benötigt keine eigene Stromquelle.
Das Display verfügt über eine I^2C-Schnittstelle [Sem]. I^2C ist ein serieller Datenbus. Er ermöglicht die Kommunikation mit mehreren, an den Bus angeschlossenen, Geräten. Jedes Gerät besitzt eine Adresse, über die dieses Geräte angesprochen wird. Dabei können sowohl Daten empfangen als auch gesendet werden. Dafür werden zwei Leitungen benötigt: Eine Datenleitung und eine Leitung für den Takt. An beide Leitungen müssen Pullup-Widerstände angeschlossen werden. In Abbildung 3.4 ist der Aufbau der Schaltung dargestellt. Der Spannungswandler ist zusammen mit den Pullup-Widerständen auf einer kleinen Lochstreifenplatine verlötet. Diese kann direkt an die Anschlusspins des Displays gesteckt werden. Der Sensorknoten selbst wird mit vier Jumper-Kabeln an der Lochstreifenplatine angeschlossen. Dadurch ist es einfach, einzelnen Komponenten des Systems auszuwechseln.

3.2.3 Schutzhülle

Die einzelnen Sensorknoten werden zusammen mit einem Display in einer Schutzhülle verbaut (siehe Abbildung 3.5). Die Hülle hat einen transparenten Deckel unter dem das Display angebracht ist. Im Innenraum der Hülle sind die Zwischenräume mit Styropor gefüllt, sodass die Sensorknoten und die Displays fixiert sind. Die Antenne der Sensorknoten wird außen an der Hülle angebracht. Dafür wird ein Verlängerungskabel benutzt, welches durch ein Loch nach außen geführt wird. Ebenfalls außen angebracht ist ein Taster, der am Sensorknoten angeschlossen ist.

3.2.4 Problematik: Transformation

Verteilte Systeme können über einen sogenanntes Steuerprogramm (engl. Scheduler) koordiniert werden. Das Steuerprogramm sorgt dafür, dass die Reihenfolge in der die Prozesse ausgeführt werden fair ist. Er stellt sicher, dass

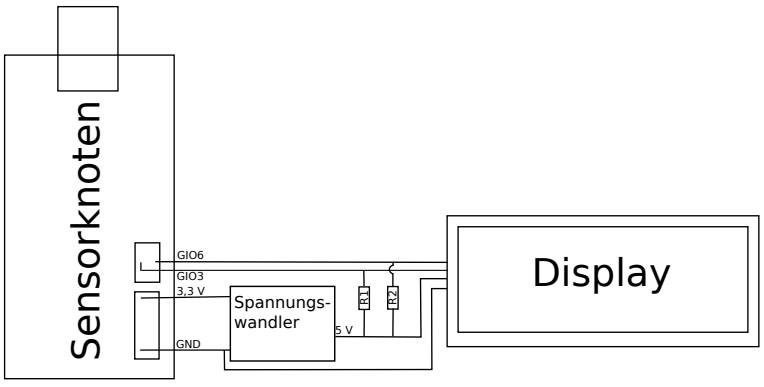


Abbildung 3.4: LCD-Schaltung

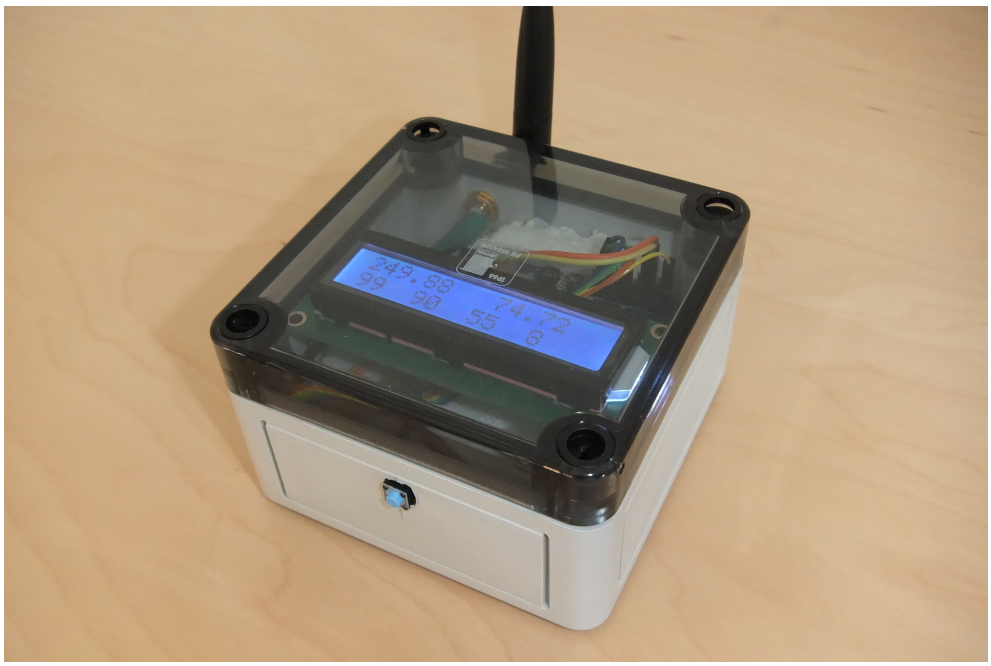


Abbildung 3.5: Schutzhülle

jeder Prozess ausgeführt wird und kein Prozess bevorzugt dran kommt. Bei einem Sensornetzwerk laufen die Programme auf den Knoten parallel. Für ein solches System ein selbststabilisierenden Algorithmus zu schreiben ist aufwendiger als für ein System mit einem fairen Steuerprogramm. Aus diesem Grund sind viele schon bekannte selbststabilisierende Algorithmen für faire Steuerprogramme geschrieben.

Sensornetzwerke kommunizieren über Nachrichtenaustausch (engl. message passing). Informationen werden mit Nachrichten über Funk übertragen. Viele selbststabilisierende Algorithmen sind für Systeme geschrieben, die Informationen über einen geteilten Speicher (engl. shared memory) austauschen. Dabei kann jeder Prozess exklusiv auf den Speicher zugreifen. So wird sichergestellt, dass der Prozess bei einem Zugriff den aktuellen Zustand der Variablen im Speicher bekommt. Bei der Kommunikation über Nachrichtenaustausch ist dies nicht gegeben. Eine Nachricht benötigt eine bestimmte Zeit, um von einem Knoten zum anderen übertragen zu werden. Empfängt ein Knoten die Daten, ist nicht sichergestellt, dass diese auch aktuell sind. Er könnte beispielsweise von einem anderen Knoten einen veralteten Wert haben und von einem Zweiten einen aktuellen. Der Knoten hat einen ungültigen Zustand vom System und kann somit nicht richtig funktionieren. Im schlimmsten Fall führt dieses Problem zu Schleifen im Netzwerk. Dabei kann sich das System nicht stabilisieren.

Um dieses Problem zu lösen, gibt es Algorithmen, die dem Nutzalgorithmus

ein Umfeld mit geteiltem Speicher simulieren. Diese Algorithmen werden auch als Transformationsalgorithmen bezeichnet. Diese simulieren den geteilten Speicher für den Nutzalgorithmus und stellen sicher, dass der Nutzalgorithmus auf aktuelle Daten zugreifen kann. Der hier verwendete Transformationsalgorithmus wird in Kapitel 6 vorgestellt und simuliert sowohl einen faires Steuerprogramm, als auch den Zugriff per geteilten Speicher.

4 Anforderungsanalyse

Bevor in Kapitel 5 auf den Entwurf des selbststabilisierenden Systems eingegangen wird, werden in diesem Kapitel zunächst die Anforderungen analysiert.

Die erste Anforderung ist, dass das System selbststabilisierend in Bezug auf den Nutzalgorithmus ist. Das schließt Fehler ein, die bei der Funkübertragung auftreten, durch das Entfernen oder Hinzufügen von Sensorknoten zum Sensornetzwerk entstehen, oder die die Programmvariablen verändern. Fehler die den Programmspeicher und damit die Programmierung der Sensorknoten beeinflussen oder Fehler, durch welche die Hardware dauerhaft beeinträchtigt ist werden hier ausgeschlossen.

Eine weitere Anforderung ist die Darstellung des selbststabilisierenden Mechanismus. Es soll für einen außenstehenden Betrachter möglich sein zu erkennen,, wie sich das System stabilisiert. Die Sensorknoten sollen also in der Lage sein Informationen über ihren Zustand z.B. über einen Display darzustellen. Außerdem sollte der Betrachter in der Lage sein das System in einen nicht stabilen Zustand zu versetzen, sodass er den selbststabilisierenden Mechanismus betrachten kann. Die Interaktion mit dem Sensornetzwerk sollte einfach und nachvollziehbar sein.
Die einzelnen Sensorknoten sollten von einer Hülle geschützt werden, sodass eine sichere Benutzung garantiert ist.

Stabilisiert sich das System, so soll durch Testfälle quantifiziert werden, ob das System selbststabilisierend ist und welche Unterschiede es zwischen der erwarteten Zeit bis zur Stabilisierung des Systems und den tatsächlichen Zeit gibt. Dazu müssen Testszenarien entwickelt werden, das System in einen instabilen Zustand versetzen und Messungen vornehmen, ob und in welcher Zeit sich das System stabilisiert.

5 Entwurf des selbststabilisierenden Algorithmus

In diesem Kapitel wird der Entwurf des Algorithmus genauer betrachtet. Dazu wird zunächst auf den Nutzalgorithmus eingegangen, der den Durchschnitt der Lichtwerte berechnet. Es folgt der Beweis, dass der Algorithmus auch tatsächlich selbststabilisierend ist und anschließend wird die Übertragung auf das Sensornetzwerk erläutert.

5.1 Entwicklung des selbststabilisierenden Algorithmus

Da das aufgebaute Sensornetzwerk zu Demonstrationszwecken dienen soll, wird ein einfacher Algorithmus als Nutzalgorithmus verwendet. Der Algorithmus liest den Wert eines Helligkeitssensors aus und speichert diesen. Jeder Sensorknoten berechnet den Durchschnittswert der Helligkeitswerte aller Sensorknoten im Sensornetzwerk. Das Prädikat P unter welchem der Algorithmus stabil ist lautet: Alle Sensorknoten haben den durchschnittlichen Helligkeitswert. Um den Durchschnittswert zu berechnen muss jeder Knoten auf die Helligkeitswerte der Anderen zugreifen. Dazu müssen die Sensorknoten miteinander kommunizieren. Außerdem kann einem Sensorknoten ein neuer Helligkeitswert zugewiesen werden indem ein Knopf an dem Sensorknoten gedrückt wird. Durch Drücken des Tasters wird der alte Helligkeitswert von dem neuen überschrieben und das System muss sich mit dem neuen Helligkeitswert stabilisieren. Der aktuelle Helligkeitswert und Durchschnittswert wird über eine Anzeige am Sensorknoten ausgegeben.
In Abbildung 5.1 ist der Ablauf des Algorithmus schematisch dargestellt und soll zur Veranschaulichung dienen. Im Folgenden wird ein Sensornetzwerk S betrachtet. Bei der Initialisierung eines Sensorknotens $S_i \in S$ wird der Wert des Helligkeitssensors ausgelesen und in der Variable h_i gespeichert. Der Durchschnittswert wird mit der Formel 5.1 berechnet.

$$d_i = \frac{\sum_{i=0}^{n} h_i}{n}, n = |P| \tag{5.1}$$

Die Helligkeitswerte h_i werden mit den anderen Knoten über den im vorherigen Kapitel beschriebenen Transformations-Algorithmus synchronisiert.
Wird ein Knopf an dem Sensorknoten betätigt, so wird der aktuelle Wert des

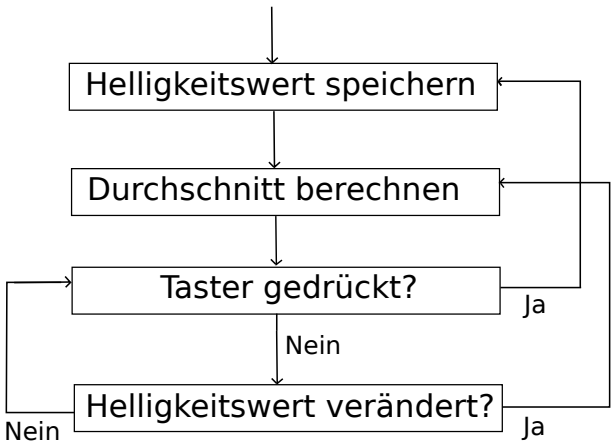


Abbildung 5.1: Ablaufplan

Helligkeitssensors in der Variable h_i gespeichert. Dadurch ist es möglich den Wert von h_i zu beeinflussen. Der Nutzalgorithmus ist für ein Umfeld mit geteiltem Speicher entworfen. Ein Sensorknoten muss auf die Variablen aller anderen Sensorknoten zugreifen können.

5.2 Beweis der Selbststabilisierung

Der Nutzalgorithmus gilt als stabil, wenn alle Sensorknoten den Durchschnittswert der Helligkeitswerte richtig berechnet haben. Der Nutzalgorithmus läuft unter einem fairen Steuerprogramm. Damit ist sichergestellt, dass jeder Sensorknoten P_i seinen Quelltext in endlich vielen Schritten ausführen kann. Wenn jeder Sensorknoten den Nutzalgorithmus einmal ausgeführt hat und sich die Helligkeitswerte währenddessen nicht geändert haben, ist der Nutzalgorithmus stabil. Daraus folgt, dass der Nutzalgorithmus in endlich vielen Schritten stabil ist. Wenn der Nutzalgorithmus stabil ist, bleibt er stabil, da sich der Durchschnittswert nicht mehr ändert.

5.3 Übertragung des Algorithmus auf das Sensornetzwerk

Der Nutzalgorithmus ist für ein Umfeld geschrieben, welches Informationen mithilfe von geteiltem Speicher austauscht. Das Sensornetzwerk, auf dem der Algorithmus zum Einsatz kommt, tauscht Informationen per Funk-Nachrichten aus. Aus diesem Grund wird, wie schon in Abschnitt 3.2.4 beschrieben, ein Transformationsalgorithmus benötigt. Dieser Transformationsalgorithmus stellt ebenfalls einen faires Steuerprogramm bereit.
Der Nutzalgorithmus setzt voraus, dass alle Sensorknoten untereinander kommunizieren können. Da nicht unbedingt alle Sensorknoten benachbart sind, ist eine Weiterleitung von Nachrichten erforderlich. Diese Weiterleitung wird im

Allgemeinen als Routing bezeichnet.

Im folgenden Kapitel werden ein Routing-Algorithmus und ein Transformationsalgorithmus beschrieben.

6 Kommunikation und Transformation

Im ersten Teil dieses Kapitels wird der Algorithmus vorgestellt, der die Kommunikation zwischen den Sensorknoten sicherstellt. Der zweite Teil beschreibt den Transformationsalgorithmus, der dem Nutzalgorithmus das Umfeld bietet, in dem er selbststabilisierend ausgeführt werden kann.

6.1 Kommunikation

Damit sich der Nutzalgorithmus stabilisieren kann, muss jeder Knoten mit jedem Anderen kommunizieren können. Da der Funksender der Sensorknoten nur eine begrenzte Reichweite hat, kann es sein, dass nicht alle Knoten direkt miteinander kommunizieren können. Nehmen wir an das Sensornetzwerk besteht aus drei Knoten: A, B und C. Nehmen wir weiterhin an Knoten A kann nur mit Knoten B kommunizieren und Knoten B nur mit C und umgekehrt. Die Knoten sind in Abbildung 9.2 dargestellt. Die Linien in der Abbildung stellen die Kommunikationsmöglichkeiten dar. Knoten A kann keine Nachricht direkt zu Knoten C schicken. Wenn die Knoten in der Lage sind Nachrichten weiterzuleiten, kann Knoten A die Nachricht an Knoten B schicken und dieser leitet die Nachricht an Knoten C weiter. Diese Weiterleitung wird im Folgenden als Routing bezeichnet. Damit Knoten A die Nachricht über Knoten B zu C schicken kann, muss Knoten A wissen, dass er Knoten C über B erreichen kann. Dafür gibt es Routing-Tabellen. In diesen stehen Informationen an wen Nachrichten weitergeleitet werden müssen, damit sie beim Empfänger ankommen.

Um diese Routing-Tabellen aufzubauen, wird in dieser Arbeit ein selbststabilisierender Algorithmus, der die Knoten in Form eines aufspannenden Baumes anordnet, benutzt.

Im Folgenden wird zuerst beschrieben, wie der aufspannende Baum aufgebaut wird und anschließend, wie mit seiner Hilfe die Nachrichten geroutet werden.

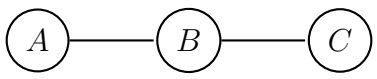


Abbildung 6.1: Routing

6.1.1 Selbststabilisierender aufspannender Baum

Zuerst soll kurz erläutert werden, was ein aufspannender Baum ist. Der in der Graphentheorie auch als Spannbaum bezeichnete aufspannende Baum ist ein zusammenhängender, kreisfreier, ungerichteter Graph, der alle Knoten beinhaltet. In unserem Fall, sind die Knoten des Graphen die Sensorknoten. In einem Baum wird der oberste Knoten als Wurzelknoten bezeichnet. Ein Baum hat genau einen Wurzelknoten. Der Knoten, der in Richtung des Wurzelknotens über einem Knoten liegt wird als dessen Vaterknoten bezeichnet.

Es gibt viele selbststabilisierende Algorithmen, die eine Netzwerkstruktur als aufspannenden Baum aufbauen. Den meisten Algorithmen liegt ein Algorithmus zur Wahl des Leitknotens zu Grunde. Dazu wird der Knoten mit der kleinsten ID (eindeutige Identifikationsnummer eines Sensorknotens) als Leitknoten gewählt. Dieses Verfahren ist noch nicht selbststabilisierend, da eine falsche ID (z.B. durch einen Netzwerkfehler entstanden) das System in einen nicht stabilen Zustand versetzt. Um dieses Problem zu lösen gibt es mehrere Ansätze. In den Artikeln [AV91] und [BAV91] wird das System zurückgesetzt, wenn ein solcher Fehler entdeckt wird. In anderen Ansätzen dürfen vorher festgelegte Grenzen (z.B. die Netzwerkgröße) nicht überschritten werden.
Der hier gewählte Ansatz aus dem Artikel [AB97], kommt völlig ohne diese Annahmen aus. Wie oben erwähnt wird auch in diesem Algorithmus der Knoten mit der kleinsten ID als Leitknoten gewählt. Um das Problem von falschen IDs zu lösen wird ein Verfahren angewandt, welches in dem Artikel als „Stromversorgung“ (engl. power supply) bezeichnet wird. Die Idee besteht darin, dass ein echter Leitknoten eine Energiequelle wird, die in regelmäßigen Intervallen „Energie“ - in Form von Nachrichten - verteilt. Die Energie fließt zu den anderen Knoten. Da Knoten mit falscher ID keine Energie erzeugen, können diese von echten IDs unterschieden werden.

Um den Algorithmus besser verstehen zu können, soll die Vorgehensweise an einem kleinen Beispiel erläutert werden. In Teil a von Abbildung 6.3 ist ein Netzwerk abgebildet. Diese Netzwerk besteht aus drei Sensorknoten mit den IDs $2, 3$ und 4. Die gestrichelten Linien zwischen den Knoten stellen dar, welche Knoten untereinander kommunizieren können. In diesem Fall können alle Knoten untereinander kommunizieren. In diesem Netzwerk würde der Knoten mit der ID 2 als Wurzelknoten ausgewählt. Die Knoten 3 und 4 würden, wie in Teil (b) abgebildet, Knoten 2 als ihren Vaterknoten wählen und damit wäre Knoten 2 der Wurzelknoten. Wir fügen nun einen weiteren Knoten mit der ID 1 hinzu. Dieser soll nur mit den Knoten 3 und 4 kommunizieren können. Wie in Teil (c) der Abbildung zu sehen ist, würden die Knoten 3 und 4 diesen

als ihren neuen Vaterknoten auswählen. Der Knoten 2 könnte den Knoten 1 nicht als Vaterknoten wählen, da er nicht in Reichweite ist und nicht mit ihm kommunizieren kann. Wie in Teil (d) gezeigt, würde der Knoten 2 entweder Knoten 3 oder Knoten 4 als Vaterknoten wählen. Welchen von beiden gewählt wird, entscheidet er anhand ihres Wurzelknotens und der Distanz zu diesem. In diesem Beispiel wären beide Knoten gleichwertig und so entscheidet, welcher Knoten sich zuerst bei Knoten 2 meldet.

6.1.2 Routing

Nachdem der in Abschnitt 6.1.1 beschriebene Algorithmus eine Netzwerkstruktur in Form eines aufspannenden Baumes erzeugt, muss sichergestellt werden, dass eine Nachricht zum Empfänger transportiert wird. Dafür wurde ein einfacher Routing-Algorithmus entworfen.

Jeder Knoten pflegt eine Routing-Tabelle, in der er die ID eines Knoten zusammen mit der ID eines Nachbarknoten speichert. Der Nachbarknoten ist derjenige, über den die Nachricht zum Empfänger gelangt. Um diese Routing-Tabelle zu füllen, sendet jeder Knoten in regelmäßigen Intervallen eine Nachricht an seinen Vaterknoten. Der Vaterknoten nimmt den Knoten in seiner Routing-Tabelle auf und sendet die Nachricht an seinen Vaterknoten weiter. Dieser Prozess wiederholt sich, bis der Wurzelknoten erreicht ist. Haben alle Knoten eine Nachricht gesendet, so kann mit Hilfe der Routing-Tabellen von dem Wurzelknoten zu jedem anderen Knoten eine Nachricht geschickt werden. Das eigentliche Routing der Pakete ist nun sehr einfach. Will ein Knoten eine Nachricht senden, so überprüft er in seiner Routing-Tabelle, wohin die Nachricht gesendet werden muss. Findet er einen Eintrag, so schickt er die Nachricht an den eingetragenen Knoten. Findet er keinen Eintrag, so schickt er die Nachricht an seinen Vaterknoten. Dieses Verfahren wollen wir anhand eines Beispiels kurz erläutern. In Abbildung 6.1.2 sehen wir ein Netzwerk mit vier Knoten. Wir nehmen an Knoten D möchte eine Nachricht an Knoten C schicken. Er überprüft seine Routing-Tabelle, Knoten C ist dort aber nicht eingetragen, also schickt er die Nachricht an seinen Vaterknoten B. Dieser kann den Knoten C ebenfalls nicht in seiner Routing-Tabelle finden und schickt die Nachricht zu seinem Vaterknoten A. Der Knoten A findet den Eintrag in seiner Routing-Tabelle und sendet die Nachricht direkt zu C weiter. Knoten C erkennt, dass die Nachricht an ihn adressiert ist und nimmt diese entgegen.

Im schlechtesten Fall, wird die Nachricht so bist zum Wurzelknoten geleitet, der sie dann zu dem entsprechenden Knoten weiterleiten kann.

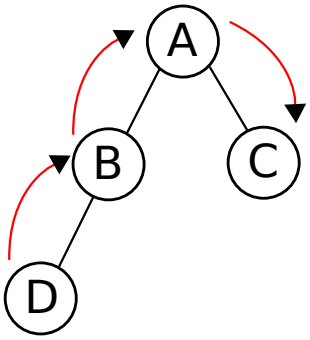
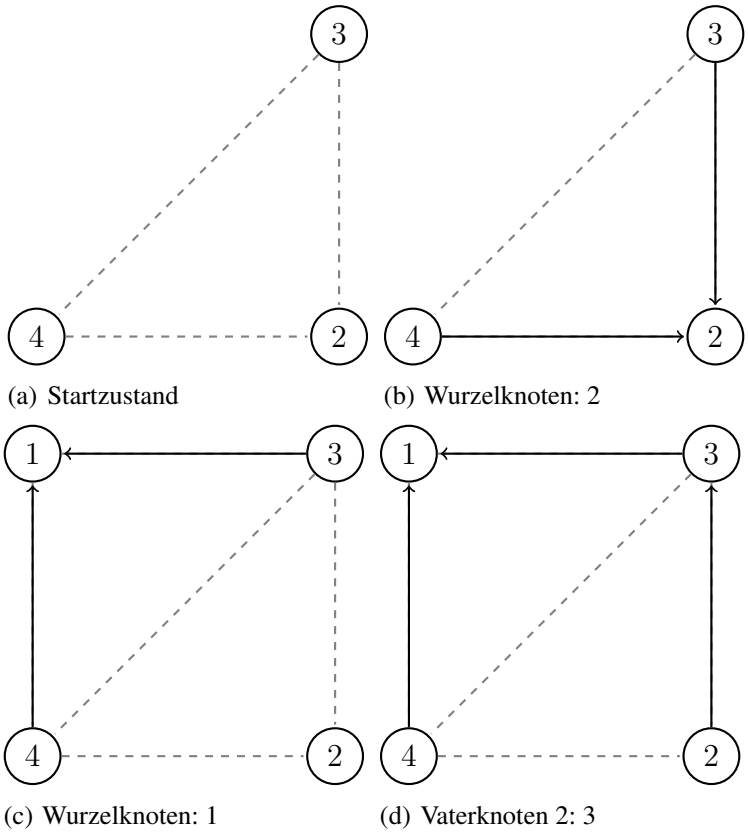


Abbildung 6.2: Routing



(a) Startzustand (b) Wurzelknoten: 2

(c) Wurzelknoten: 1 (d) Vaterknoten 2: 3

Abbildung 6.3: Beispiel zum aufspannenden Baum

6.2 Transformationen

Wie schon in Abschnitt 3.2.4 beschrieben, wird ein Transformationsalgorithmus benötigt.
Der in dieser Arbeit benutzte Algorithmus stammt von Masaaki Mizuno und Hirotsugu Kakugawa aus [MK96]. Dieser Algorithmus wurde für den Einsatz in verteilten Systemen entwickelt und bietet sich daher für den Einsatz in einem Sensornetzwerk an. Wie in Abbildung 6.4 dargestellt, wird der Nutzalgorithmus in den Transformationsalgorithmus eingebettet.
Der Transformationsalgorithmus stellt dem Nutzalgorithmus den Zugriff auf simulierten geteilten Speicher (engl. shared memory) bereit. Geteilter Speicher wird zum Informationsaustausch eingesetzt, wenn mehrere Prozesse auf einem System untereinander Daten austauschen sollen. Jeder Prozess besitzt einen Teil des Speichers, auf den er schreibend zugreifen kann. Auf die anderen Bereiche kann er nur lesend zugreifen. Der Zugriff auf den Speicher geschieht in einem atomaren Schritt. Der Schritt besteht aus dem Lesen des Speichers der anderen Prozesse und anschließend aus dem Schreiben in den eigenen Speicherbereich. Im Folgenden wird der Ablauf des Algorithmus erläutert.

Der Algorithmus Obwohl der Algorithmus auch auf anderen verteilten Systemen eingesetzt werden kann, werden wir den Algorithmus im Folgenden anhand eines Sensornetzwerkes vorstellen. Jeder Sensorknoten besitzt eine Variable q, die für den Nutzalgorithmus seinen Bereich im geteilten Speicher darstellt. Für alle anderen Knoten hat er je eine Variable, die den von diesem Knoten empfangene Wert enthält. Außerdem verfügt jeder Knoten über zwei Listen mit Knoten-IDs. Eine der beiden Listen ist die Lese-Liste. In ihr befinden sich Knoten-IDs der Knoten, die auf den Registerwert des Knotens lesend zugreifen müssen. Die andere List ist die Schreib-Liste. Von den Knoten, deren Knoten-IDs auf der Schreib-Liste sind, benötigt ein Sensorknoten eine Bestätigung, bevor er seine Variable q verändern kann.
An die Knoten aus der Lese-Liste werden in bestimmten Abständen Nachrichten mit dem aktuellen Wert q eines Sensorknotens gesendet. Die Knoten empfangen diese Nachrichten und speichern sie lokal ab. Will ein Knoten den

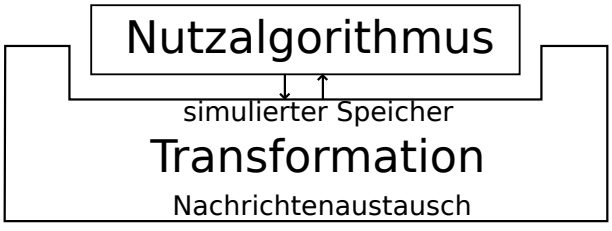


Abbildung 6.4: Transformation

Wert von q ändern ohne, dass er die Werte der anderen Knoten dazu lesen muss, schreibt er den Wert direkt in die Variable q. Will er seine Variable q in Abhängigkeit der Variablen der anderen Knoten verändern, so muss er erst eine Erlaubnis einholen. Durch dieses Vorgehen wird sichergestellt, dass der Knoten auf die aktuellen Werte der anderen Knoten zugreift. Diese Erlaubnis wird bei den anderen Knoten angefragt und kann von diesen angenommen oder abgelehnt werden. Hat ein Knoten die Erlaubnis aller Knoten aus seiner Schreib-Liste, kann er seinen Wert q verändern.
Alle Nachrichten werden mit einem Zeitstempel versehen. Mithilfe des Zeitstempels können veraltete Nachrichten identifiziert werden.

7 Implementierung

Dieses Kapitel befasst sich mit der Implementierung der in den Vorangegangenen Kapiteln beschriebenen Algorithmen. Zuerst wird auf die Programmierung mit dem Betriebssystem Contiki-OS eingegangen. Anschließend wird der Aufbau des gesamten Programms dargestellt, um eine Übersicht über die, im Folgenden beschriebenen, Teile des Programms zu geben.

7.1 Contiki-OS

Wie schon in Abschnitt 3.2.1 erwähnt, wird auf den Sensorknoten das Betriebssystem Contiki-OS [Con] eingesetzt. Contiki-OS wird verwendet, da es ein ereignisgesteuertes Betriebssystem ist und es einen sehr guten Simulator gibt. Das Simulationsprogramm heißt Cooja und ist in der Lage Sensornetzwerke zu simulieren. Bei der Programmierung von Programmen für Sensornetzwerke kann ein Simulator sehr hilfreich sein, da das Verhalten eines Sensornetzwerkes im Simulator genauestens beobachtet werden kann.
Contiki-OS ist in der Programmiersprache C geschrieben und auch die Programme für Contiki-OS werden in C geschrieben. Ein Programm besteht aus einem oder mehreren Prozessen. Prozesse können mit einer Reihe von Befehlen gesteuert werden. Ein Prozess kann beispielsweise angehalten werden, bis ein bestimmtes Ereignis eintritt. Angehaltene Prozesse können, von einem anderen Prozess, durch das Senden von Ereignissen aufgeweckt werden. Wird ein Prozess angehalten, so gehen alle nicht statischen Variablen verloren.
Für die Kommunikation zwischen den Sensorknoten werden die Netzwerkprotokolle Broadcast und Runicast des Rime-Stacks verwendet. Beide Protokolle benutzen kein Routing, was bedeutet, dass Nachrichten nur zu Sensorknoten in Funkreichweite geschickt werden können.

7.2 Aufbau

Der komplette Algorithmus setzt sich aus drei selbststabilisierenden Teilalgorithmen zusammen: dem Routing-, dem Transformations- und dem Nutzalgorithmus (siehe Abbildung 7.1). Damit der Nutzalgorithmus stabilisieren kann, muss zuerst der Transformationsalgorithmus stabilisieren. Können sich alle Sensorknoten untereinander über Funk erreichen, so kann der Transforma-

tionsalgorithmus ohne den Routingalgorithmus stabilisieren, ansonsten muss zuerst der Routingalgorithmus stabilisieren.
Das Programm besteht aus den in Abbildung 7.2 gezeigten sieben Prozessen. Alle Prozesse werden beim Anschalten der Sensorknoten gestartet und werden danach ausgeführt. Die Prozesse „SP Timeout“, „MP Timeout“, „RP Timeout“ und „Broadcast Timeout“ werden als Timeout-Prozesse bezeichnet. Ein Timeout-Prozess ist ein Prozess der in gewissen Abständen ausgeführt und den Rest der Zeit pausiert wird. Der Prozess „Broadcast Timeout“ wird beispielsweise nach dem Ausführen für eine Zeit von 20 bis 36 Sekunden pausiert, um anschließend erneut ausgeführt zu werden. Es wird kein festes Zeitintervall gewählt, um zu vermeiden, dass Sensorknoten über einen längeren Zeitraum parallel ihre Timeout-Prozesse ausführen. Da die Timeout-Prozesse Nachrichten an andere Knoten senden, könnte ein paralleles Ausführen Fehler bei der Übertragung auslösen und dadurch zu einem nicht stabilisierenden Algorithmus führen.
Zusätzlich zu den Prozessen gibt es noch zwei Callback-Funktionen. Diese Funktionen werden aufgerufen, sobald eine Nachricht eines anderen Sensorknotens eintrifft. Die Funktion „runicast_callback“ nimmt Nachrichten entgegen, die direkt an Sensorknoten gesendet werden und die Funktion „broadcast_callback“ nimmt Nachrichten entgegen, die per Broadcast an alle Knoten in Reichweite gesendet werden.
Der Quelltext ist in mehrere Dateien aufgeteilt. Die Hauptdatei heißt „node.c“ (Anhang C.2). In ihr befinden sich die Prozesse und die Callback-Funktionen. Die Datein „spanningtree.h“ (Anhang C.4) und „routing.h“ (Anhang C.6) beinhaltet den Routingalgorithmus. Der Transformationsalgorithmus befindet sich in der Datei „messagepassing.h“ (Anhang C.5). In der Datei „functions.h“ (Anhang C.3) sind Funktionen abgelegt, die z.B. zur Ausgabe auf dem Display benötigt werden. Für die Ansteuerung des Displays wird die Datei „LCD.c“ und die Datei „MSP430_SWI2C_Master.c“ benutzt.

7.3 Ansteuerung des Displays

An jedem Sensorknoten ist ein Display angeschlossen. Das Display wird wie in Abschnitt 3.2.2 beschrieben über die I^2C-Schnittstelle angesteuert. Der

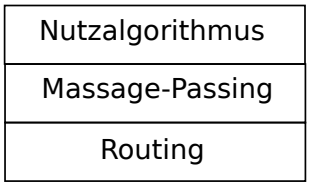


Abbildung 7.1: Schichten

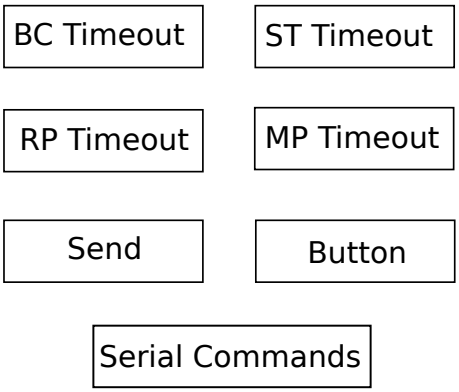


Abbildung 7.2: Prozesse

Mikrocontroller der Sensorknoten hat Anschlüsse, die mit der I^2C Funktion geschaltet werden können. Dieser Anschluss ist bei den MTM-CM5000MSP Sensorknoten schon mit einer anderen Funktion (SPI) geschaltet, um mit dem Funkmodul zu kommunizieren.
Um den Anschluss mit zwei Funktionen zu betreiben wäre ein sogenanntes Multiplexing der beiden Protokolle nötig. Multiplexing ist eine Technik bei der die beiden Protokolle abwechselnd die Leitung benutzen können. In Contiki-OS ist ein solches Multiplexing nicht vorgesehen.
Für dieses Projekt wird mithilfe einer Bibliothek das I^2C-Protokoll auf einem der anderen Anschlüsse simuliert. Das Simulieren des I^2C-Protokolls stellt keine Schwierigkeit dar, da das I^2C-Protokoll im Gegensatz zu z.B. dem UART-Protokoll kein genaues Timing voraussetzt. Die hier verwendete Bibliothek [Wu08] stammt von Texas Instruments und wurde von Randy Wu entwickelt. Das Display wird mit einer Reihe von Steuerbefehlen bedient. Eine fertige Bibliothek zum Ansteuern liegt für die Arduino-Plattform vor [DFR]. Diese wurde im Rahmen dieser Arbeit für die Verwendung mit der I^2C-Library und Contiki-OS angepasst.

7.3.1 Senden und Empfangen von Nachrichten

Zum Senden von Nachrichten wird das Rime-Protokoll von Contiki-OS verwendet. Um Nachrichten mithilfe des Rime-Protokolls empfangen zu können, werden Funktionen als Callback-Funktionen definiert. Diese Funktionen werden aufgerufen, sobald eine Nachricht an einem Sensorknoten eintrifft. In der Funktion kann die angekommene Nachricht eingelesen und behandelt werden. Das Senden von Nachrichten kann ebenfalls über den Aufruf einer Funktion geschehen. In der Implementierung wird diese Funktion nicht direkt aufgerufen, da dies zu Problemen führen kann. Werden mehrere Nachrichten direkt hintereinander versandt, so kommt es zum Verlust einiger Nachrichten, da der verwendete Nachrichtenpuffer nicht alle Nachrichten aufnehmen kann.

Um dieses Problem zu umgehen, gibt es einen Prozess, der für das Senden der Nachrichten zuständig ist und wartet, bis eine Nachricht gesendet wurde, bevor er die Nächste sendet. Es wird ein Ereignis mit der zu übertragenden Nachricht an den „Sende Prozess“ geschickt. Dieser erkennt anhand des Nachrichtentyps, ob diese Nachricht an einen oder mehrere Sensorknoten geschickt werden muss und verschickt diese anschließend. Nach jeder Nachricht wird der Prozess angehalten, bis diese verschickt wurde, dadurch gehen keine Nachrichten mehr verloren, die direkt nacheinander geschickt wurden. Der „Sende Prozess“ befindet sich in der Datei „node.h“ (Anhang C.2).

7.3.2 Finden der Nachbarknoten

Der Algorithmus zum Aufbauen eines aufspannenden Baums (Abschnitt 6.1.1) setzt voraus, dass ein Sensorknoten seine Nachbarknoten kennt. Dazu pflegt jeder Sensorknoten eine Liste mit Nachbarknoten. Der Prozess „Broadcast-Timeout“ sendet dazu Broadcast-Nachrichten. Diese Nachrichten werden von allen Sensorknoten empfangen, die sich in dessen Reichweite befinden. Bekommt ein Knoten diese Nachricht, so fügt er den Absender zu seiner Liste mit Nachbarn hinzu. Wenn sich die Struktur des Netzwerks nicht ändert, dann hat jeder Knoten, nachdem alle Knoten im Netzwerk eine Broadcast-Nachricht verschickt haben, eine vollständige Nachbar-Liste.
Knoten die nicht mehr in der Nachbarschaft sind, werden durch einen Timeout-Zähler identifiziert. Sendet ein Knoten seine Broadcast-Nachricht, so erhöht er für alle anderen Knoten in seiner Liste einen Timeout-Zähler. Bekommt er eine Nachricht von einem Knoten, so setzt er diesen Zähler wieder auf 0. Überschreitet einer der Zähler den Wert 3, so wird der entsprechende Knoten aus der Liste entfernt. Der Prozess „Broadcast-Timeout“ und die Callback-Funktion befinden sich in der Datei „node.h“ (Anhang C.2).

7.3.3 Routingalgorithmus

Der Abschnitt 6.1.1 beschreibt die Funktionsweise des Routingalgorithmus. In der Implementierung wird der Timeout-Mechanismus über den Prozess „SP-Timeout“ realisiert. Dieser Prozess sendet Nachrichten an die Nachbarknoten. Die Adressen des Vaterknotens und des Wurzelknotens werden in statischen Variablen gespeichert. Die im Algorithmus benötigten Listen werden durch Listen der Contiki-List-Library dargestellt. Empfängt der Sensorknoten eine Nachricht, die den Spanning-Tree Algorithmus betrifft, so wird die Funktion „spanningtree“ aus der Datei „spanningtree.h“ aufgerufen. Diese Funktion behandelt dann die Nachricht entsprechend des Algorithmus.

Das Routing wird wie bereits in Abschnitt 6.1.2 beschrieben durchgeführt.

Die regelmäßig gesendeten Aktualisierungsnachrichten werden vom Prozess „RP-Timeout“ gesendet. Dieser ist auch für das Löschen von Einträgen aus der Routing-Liste zuständig. Dies funktioniert ähnlich dem in Abschnitt 7.3.2 beschriebenen Timeout-Verfahren zum Löschen der Nachbarknoten. Ob eine Nachricht geroutet werden muss oder nicht wird anhand der Zieladresse, die jede Nachricht hat, festgestellt. Der Aufruf der Funktion „routing“ liefert die Adresse des Sensorknotens zurück, an die die Nachricht als nächstes geschickt werden muss.

7.3.4 Transformationsalgorithmus

Der Transformationsalgorithmus befindet sich in der Datei „messagepassing.h“ und wird mit der Funktion „messagepassing“ aufgerufen. Auch dieser Algorithmus hat einen Timeout-Mechanismus, welcher durch den Prozess „MP-Timeout“ abgebildet wird. Die beiden benötigten Listen für die Sensorknoten mit schreibendem und lesendem Zugriff werden als Contiki-Listen abgebildet. Die Listen werden am Anfang des Prozesses „MP-Timeout“ mit Adressen der Sensorknoten gefüllt. Für den in dieser Arbeit verwendeten Nutzalgorithmus werden die Adressen der anderen Knoten hier in die Lese-Liste eingetragen, da die anderen Knoten lesenden Zugriff auf die Variable des Sensorknotens haben. Die Liste mit den Knoten für den schreibenden Zugriff bleibt leer, da die Sensorknoten ihren Helligkeitswert nicht in Abhängigkeit der anderen Helligkeitswerte verändern. Sie berechnen mit den anderen Helligkeitswerten lediglich den Durchschnittswert.

7.3.5 Nutzalgorithmus

Der Nutzalgorithmus berechnet mithilfe der Funktion „average“ den Durchschnittswert der Helligkeitswerte. Die Funktion wird nach dem Empfangen einer Nachricht aufgerufen, da der Durchschnittswert sich nach Eintreffen einer Nachricht geändert haben kann. Hat dieser sich geändert, so wird der neue Wert auf dem Display ausgegeben. Es wird ebenfalls überprüft, ob sich die Adresse des Vaterknotens geändert hat und ggf. auch diese Adresse auf dem Display aktualisiert.
Der Prozess „Button_process“ überprüft, ob der Taster gedrückt wurde. Wird dieser gedrückt, so wird der aktuelle Wert des Helligkeitssensors gespeichert und danach auf dem Display ausgegeben.

7.3.6 Serielle Kommunikation

Ist ein Sensorknoten per USB mit einem Computer verbunden, so kann der Knoten während er sein Programm ausführt mit dem Computer kommunizieren.

Die am Computer eingegebenen Befehle werden vom Prozess „Serial Commands“ verarbeitet. Mithilfe der Befehle kann das Verhalten des Netzwerkes genauer analysiert werden. Folgende Befehle stehen zur Verfügung:

Info-Befehl Der Info-Befehl veranlasst den Sensorknoten Informationen über seinen aktuellen Zustand auszugeben. Es werden die Adresse des Vaterknotens und die Adresse des Wurzelknotens ausgegeben.

Ping-Befehl Mit dem Ping-Befehl wird eine geroutete Nachricht an einen Sensorknoten geschickt. Bekommt dieser die Nachricht, antwortet er mit einer Pong-Nachricht. Dadurch kann die Erreichbarkeit eines Knotens überprüft werden.

8 Darstellung

Das in dieser Arbeit entwickelte System soll zu Demonstrationszwecken genutzt werden. Einem Beobachter oder Benutzer des Systems soll den selbststabilisierenden Mechanismus nachvollziehen können. Um dies zu ermöglichen, wurden die Sensorknoten mit einem Display und einem Taster ausgestattet. In diesem Kapitel wird die Interaktion mit dem System beschrieben und anschließend ein Vorschlag unterbreitet, wie das System zur Demonstration genutzt werden kann.

8.1 Interaktion mit dem System

Sind die Sensorknoten eingeschaltet, werden auf dem Display vier Werte ausgegeben. In Abbildung 8.1 wird eine solche Ausgabe gezeigt. Das in der Abbildung mit 1 gekennzeichnete Feld ist die Adresse des Sensorknotens. Rechts daneben und mit 2 gekennzeichnet wird die Adresse des Vaterknotens angezeigt. In diesem Fall hat der Knoten keinen Vaterknoten, weswegen „0.0“ angezeigt wird. Das nächste Feld gibt den vom Knoten gespeicherten Helligkeitswert an und das letzte Feld beinhaltet den ermittelten Durchschnittswert.
Als Eingabemöglichkeit befindet sich an der Box ein Taster. Wird dieser gedrückt, so wird der Helligkeitssensor ausgelesen und der Wert auf dem Display in Feld 3 angezeigt. Der in Feld 4 angezeigte Durchschnittswert wird sofort mit dem neuen Helligkeitswert berechnet.

8.2 Demonstrationsvorschläge

Das in dieser Arbeit entworfenen System könnte z.B. auf einem Informationstag für Schülerinnen und Schüler an der Universität vorgestellt werden. Diese

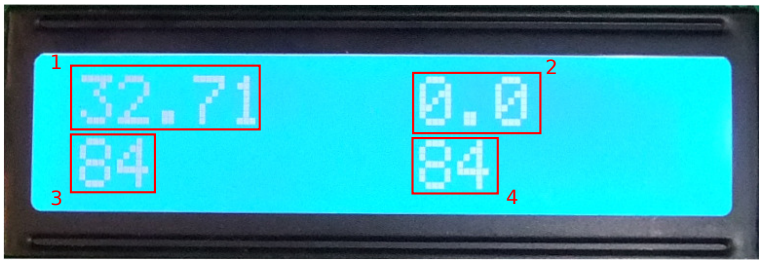


Abbildung 8.1: Ausgabe auf dem Display

Informationstage sollen ihnen einen Einblick ins Studium geben.
Interessierten Schülerinnen und Schülern sollte zuerst erklärt werden, was Sensorknoten sind und wo sie eingesetzt werden. Damit die Schülerinnen und Schüler ein Bild davon haben, wie ein Sensorknoten aussieht, kann einer aus der Schutzhülle herausgenommen werden. An dem Sensorknoten können die Bestandteile gezeigt werden. Hierbei ist es sinnvoll den Helligkeitssensor zu zeigen, da dieser im folgenden Versuchsaufbau benutzt wird.
Vom Algorithmus ist hauptsächlich der Teil, der den aufspannenden Baum aufbaut und der Nutzalgorithmus interessant. Diese beiden Teile könnten anhand einer Grafik erklärt werden.
Anschließend können Schülerinnen und Schüler sich die Sensorknoten angucken und neue Helligkeitswerte einstellen. Zum Ändern der Helligkeitswerte wäre es empfehlenswert, eine Taschenlampe oder eine andere Lichtquelle bereitzustellen. Mit dieser kann der Helligkeitssensor auf dem Sensorknoten eingeleuchtet werden und so ein höherer Messwert erzielt werden.
Die Schülerinnen und Schüler können versuchen einen möglichst hohen oder möglichst tiefen Durchschnittswert zu erzeugen. Der berechnete Durchschnittswert kann zur Überprüfung von den Schülern nachgerechnet werden.

9 Quantifiziereung der selbststabilisierenden Eigenschaften

In diesem Kapitel werden Testszenarien beschrieben, die zur Validierung der selbststabilisierenden Eigenschaften dienen. Die durch die Testszenarien erhobenen Daten werden im Anschluss ausgewertet. Zu jedem Testszenario wird die erwartete Dauer, bis der jeweilige Algorithmus einen stabilen Zustand erreicht, bestimmt. Dieser Wert kann dann mit der gemessenen Zeit verglichen werden. Die ersten drei Testszenarien beziehen sich nicht auf einen der selbststabilisierenden Teilalgorithmen, sondern überprüfen das Erkennen der Nachbarknoten. Diese Tests wurden durchgeführt, um Fehler bei der Erkennung von Nachbarknoten auszuschließen. Diese Fehler würden ansonsten das stabilisieren der selbststabilisierenden Algorithmen beeinflussen.

9.1 Testszenarien zur Quantifizierung

Die Selbststabilisierung des Algorithmus soll mit einigen Testszenarien überprüft werden. Dazu werden auf den Sensorknoten Funktionen aufgerufen, welche Messungen vornehmen. Die zu testenden Sensorknoten müssen dazu über die USB-Schnittstelle mit einem Computer verbunden sein. Die Funktionen können dann durch Eingabe von Befehlen gestartet werden. Genauere Anweisungen, wie diese Befehle eingegeben werden, sind im Abschnitt B.2 zu finden. Für diese Tests wurde eine separate Datei „node_tests.c“ angelegt, die für die Testläufe extra auf die Sensorknoten übertragen werden muss. Die Schritte die dafür nötig sind, werden in Abschnitt B.1.2 genauer erläutert.

In den Testszenarien wird jeweils gemessen, wie lange das Sensornetzwerk benötigt damit auf einem Knoten ein bestimmtes Ereignis eintritt. In dieser Zeitspanne werden je nach Test das Ausführen von Timeout-Prozessen und das Eintreffen bestimmter Nachrichten gemessen. Diese Messwerte können dann mit den erwarteten Werten verglichen werden.

Betrachten wir nun das erste Testszenario. Dieses Testszenario überprüft das Finden von Nachbarknoten. Die Funktion wird mit dem Befehl „tbc1“ gefolgt von einer Test-Adresse aufgerufen und soll hier exemplarisch erläutert werden. In Abbildung 9.1 ist ein Ausschnitt der Funktion abgebildet. In Zeile 1 und 2 werden die Zähler-Variablen auf den Wert 0 gesetzt. Diese statischen Variablen

werden bei Eintreffen einer Nachricht oder Auslösen eines Timeout-Prozesses hochgezählt. In Zeile 3 wird die Variable zum Zeitmessen initialisiert. In der fünften Zeile wird das Eintreffen eines Ereignisses überprüft. In diesem Testszenario wird überprüft, ob eine Adresse eines bestimmten Knotens in der Nachbar-Liste ist. Dazu wird die Funktion „is_alive“ mit einer vorher festgelegten Test-Adresse aufgerufen. Die Schleife in Zeile 5 wird solange ausgeführt, bis diese Adresse in der Nachbar-Liste auftaucht. In der Schleife wird „PROCESS_PAUSE“ aufgerufen. Dieser Aufruf pausiert den aktuellen Prozess und gibt den anderen Prozessen die Möglichkeit ausgeführt zu werden. Dies ist nötig, da dieser Prozess ansonsten die anderen blockieren würde. Ist der Test beendet werden die gesammelten Daten ausgegeben.

9.1.1 Nachbarerkennung

Als erstes soll überprüft werden wie lange es dauert, bis ein Sensorknoten einen anderen Sensorknoten als Nachbarknoten findet. Damit ein Knoten einen anderen Knoten als Nachbarn erkennt, muss er eine Broadcast-Nachricht von diesem empfangen. Jeder Knoten sendet Broadcast-Nachrichten durch den Timeout-Prozess „Broadcast_timeout“. Dieser wird alle 20 bis 36 Sekunden ausgeführt (siehe Tabelle A.1). Das Erkennen eines neuen Knotens sollte also nicht länger als 36 Sekunden dauern.
Zum Überprüfen der Nachbarerkennung wurden zwei Testszenarien durchgeführt, die im Folgenden beschrieben werden.

Testszenario 1 Für dieses erste Testszenario werden zwei Sensorknoten benutzt. Den ersten Sensorknoten bezeichnen wir als A und den zweiten als B. Auf Knoten A wird gemessen, wie viele Broadcast-Nachrichten eintreffen und wie lange es dauert, bis der Nachbarknoten gefunden wird. Für diese Messung wird die in Abbildung 9.1 gezeigte Funktion benutzt. Auf Knoten B soll die Anzahl der Broadcast-Timeouts gemessen werden. Knoten B gibt diese auf

```
1          bc_timeouts = 0;
           bc_messages = 0;
3          starttime = clock_seconds();

5          while(is_alive(&test_addr)) {
              PROCESS_PAUSE();
7          }
           printf("Knotenerkennungstest abgeschlossen.
               st_t: %d, st_m: %d, time %d\n",
              bc_timeouts, bc_messages, (clock_seconds
              () - starttime));
```

Abbildung 9.1: Unterprogramm zum Testen

dem Display aus, wo sie dann abgelesen werden müssen. Sobald auf Knoten A die Messung gestartet wurde, wird Knoten B angeschaltet.

Die Messergebnisse können in Abbildung A.2 im Anhang unter A.2 gefunden werden. Wie zu erwarten empfängt der Knoten A bei allen Messungen genau eine Broadcast-Nachricht und bei Knoten B wird genau einmal der Timeout ausgelöst. Die Dauer der Messungen liegen innerhalb des erwarteten Bereichs von 20 bis 36 Sekunden.

Testszenario 2 In diesem Testszenario soll der unter Testszenario 1 beschriebenen Test mit fünf Sensorknoten wiederholt werden. Die Baumstruktur wird in Abbildung 9.3 gezeigt. Es wird gemessen, wie lange es dauert bis Knoten A die Nachbarn B, C, D und E gefunden hat. Die Knoten befinden sich alle in Reichweite von Knoten A. Da Knoten die Broadcast-Nachrichten an alle Knoten in Reichweite gleichzeitig senden, wird erwartet, dass die Laufzeit dieses Tests ebenfalls zwischen 20 und 36 Sekunden liegt.

Wie in Tabelle A.3 zu sehen ist, werden auch bei diesem Test die Erwartungen erfüllt.

9.1.2 Löschen der Nachbarknoten

Im vorangegangenen Abschnitt wurde gemessen, wie lange es dauert einen Nachbarknoten zu erkennen. Nun wird überprüft, wie lange es dauert zu erkennen, dass ein Knoten nicht mehr erreicht werden kann.

Testszenario 3 Wie schon im vorherigen Abschnitt werden zunächst zwei Sensorknoten A und B benutzt. Auf Knoten A werden wieder die Broadcast-Timeouts gezählt und die Laufzeit des Tests gemessen. Sobald Knoten A Knoten B gefunden hat, wird der Test gestartet. Die Messung auf Knoten A wird ausgelöst und Knoten B wird ausgeschaltet.
Es werden Laufzeiten zwischen 60 und 108 erwartet, da ein Knoten einen Anderen löscht sobald er innerhalb von drei Broadcast-Tiemouts keine Nachricht von ihm bekommen hat.

Die in Abbildung A.4 gezeigten Messergebnisse stimmen mit den erwarteten Messergebnissen überein. Es wurde erwartet, dass der Broadcast-Timeout-Prozess genau drei Mal durchlaufen wird bevor der Knoten aus der Nachbar-Liste entfernt wird. Die Messergebnisse bestätigen dieses im Test mit zwei und im Test mit fünf (A.5) Sensorknoten. Die durchschnittlich Zeit, die ein

Sensorknoten benötigt um zu erkennen, dass ein Anderer nicht mehr erreichbar ist, liegt zwischen 60 und 108 Sekunden.

9.1.3 Finden des Wurzelknoten

In diesen Testszenarien wird untersucht, wie lange es dauert bis ein Knoten den Wurzelknoten erkannt und sich im aufspannenden Baum eingeordnet hat.

Testszenario 4 Zuerst wird der Test mit zwei Sensorknoten durchgeführt. Sensorknoten A wird dabei die Rolle des Wurzelknotens einnehmen. Auf Knoten A wird die Anzahl der Timeouts des Prozesses „Spanningtree_timeout“, die eingehenden Nachrichten und die Dauer, bis ein Vaterknoten akzeptiert wurde, gemessen. Auf Knoten B werden ebenfalls die Timeouts und die Nachrichten gemessen.
Betrachten wir nun welche Messergebnisse erwartet werden. Knoten A muss sich selbst als Wurzelknoten erkennen. Dazu ist es nötig, dass er eine Nachricht von Knoten B bekommt. Knoten B muss seinen Timeout-Prozess also einmal betreten. Danach muss Knoten B drei Nachrichten von A erhalten, um diesen als Vater- und Wurzelknoten anzunehmen. Da die Timeout-Prozesse wie aus Tabelle A.1 ersichtlich alle 15 bis 20 Sekunden ausgeführt werden, wird bei vier Durchläufen eine Zeit zwischen 60 und 80 Sekunden erwartet.

Die Messergebnisse sind in Tabelle A.6 zu finden. In Abbildung 9.4 Diagramm (a) werden die gemessenen Zeiten dargestellt. Die beiden Linien stellen den oberen und unteren Erwartungswert dar. Wie zu erkennen ist, liegen die meisten Messwerte oberhalb dieses Erwartungsbereichs.

Testszenario 5 In diesem Szenario kann nicht jeder Knoten mit jedem Anderen kommunizieren. Die benutzten drei Knoten sind so angeordnet, dass sich die in Abbildung 9.2 gezeigte Baumstruktur bildet. Es werden die selben Daten wie im vorangegangenen Test gemessen. Auf dem mittleren Knoten C werden keine Daten gemessen. Die Knoten werden alle gleichzeitig eingeschaltet und

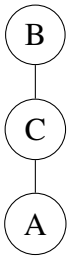


Abbildung 9.2: Baumstruktur 1

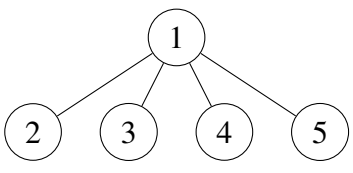


Abbildung 9.3: Baumstruktur 2

die Messung auf Knoten A gestartet. Die Messung läuft solange, bis Knoten A Knoten B als Wurzelknoten angenommen hat.
Erwartet wird eine Laufzeit zwischen 120 und 180 Sekunden, da zuerst Knoten C den Wurzelknoten B als Vaterknoten annehmen muss, bevor Knoten A über Knoten C Knoten B als Wurzelknoten findet. Zwar kann Knoten A schon vorher Knoten C als Vaterknoten wählen, die Messung wird aber durchgeführt, bis Knoten B als Wurzelknoten erkannt wird.

Die Messergebnisse können in Tabelle A.7 nachgelesen werden. Die Dauer des Tests wird im Diagramm 9.4 (b) dargestellt. Wie zu erkennen ist liegen die gemessenen Werte teilweise deutlich über den erwarteten Werten.

9.1.4 Transformations- und Nutzalgorithmus

In diesem Abschnitt werden Messungen zum Transformations- und Nutzalgorithmus durchgeführt.

Testszenario 6 Im diesem Testszenario werden zwei Sensorknoten verwendet. Auf dem mit A gekennzeichneten Knoten werden die Timeouts des Prozesses „Messagepassing-Timeout“ und die Laufzeit des Tests gemessen. Sind die berechneten Durchschnittswerte auf beiden Knoten stabil, so wird auf Knoten B ein neuer Helligkeitswert eingestellt. Der Test wird beendet, sobald beide Durchschnittswerte wieder stabil sind.
Da der Prozess „Messagepassing-Timeout“ alle 8 bis 13 Sekunden ausgeführt wird, sollte der Test 8 bis 13 Sekunden dauern. Da der Test erst gestartet wird, wenn der Timeout-Prozess schon mehrere male ausgeführt wurde, kann dieser auch sofort ausgeführt werden. Aus diesem Grund kann die Laufzeit auch unter 8 Sekunden liegen. Es werden damit Werte zwischen 0 und 13 Sekunden angenommen.

Die in Tabelle A.8 abgebildeten Messwerte liegen bis auf zwei Ausnahmen (Messwerte 3 und 9) innerhalb des erwarteten Bereichs. Die Dauer des Tests wird in Diagramm 9.6 dargestellt.

Testszenario 7 Nun wollen wir einen Test mit fünf Sensorknoten ausführen. In diesem Testszenario kann die Laufzeit nicht mithilfe einer Funktion auf den Sensorknoten gemessen werden, da die Funktion nur die Stabilisierung auf einem Sensorknoten messen kann. Deswegen wird die Zeit manuell mit einer Stoppuhr gemessen. Die Durchschnittswerte werden von den Displays abgelesen und überprüft. Die fünf Knoten können bei diesem Test alle direkt miteinander kommunizieren. Im Gegensatz zum vorherigen Test wird diesmal

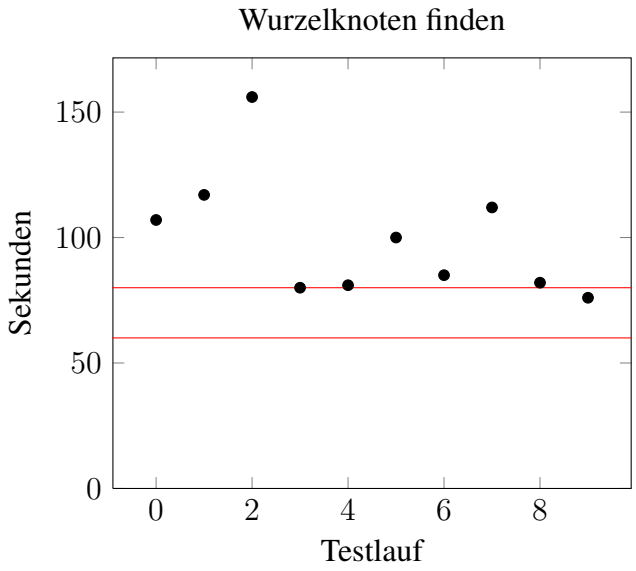


Abbildung 9.4: Wurzelknoten finden 1

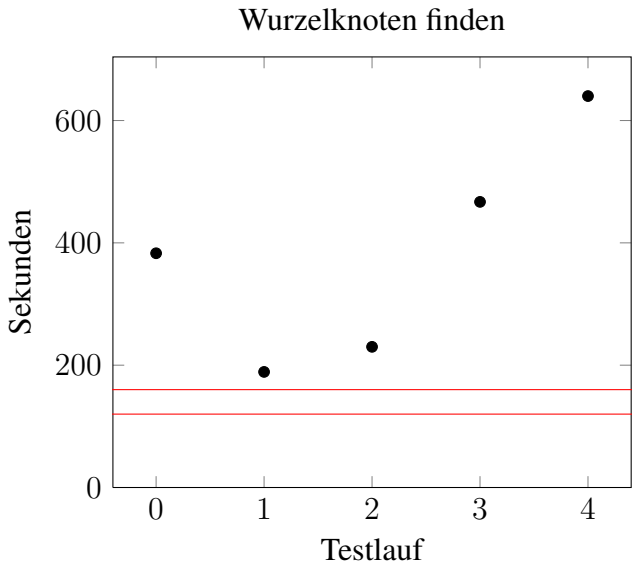


Abbildung 9.5: Wurzelknoten finden 2

die Zeit vom Starten der Sensorknoten bis zu dem Zeitpunkt, an dem die Durchschnittswerte stabil sind, gemessen.
Bei diesem Testszenario wird erwartet, dass die Durchschnittswerte innerhalb von 28 bis 49 Sekunden stabil sind, da die Sensorknoten zwischen 20 und 36 Sekunden benötigen um sich gegenseitig zu finden und den Prozess „Messagepassing-Timeout“ auf jedem Knoten einmal ausführen werden muss, damit sein Helligkeitswert an alle anderen Knoten verteilt wird.

Die Messwerte sind in Tabelle A.9 abgebildet und in Diagramm 9.7 sind die gemessenen Laufzeiten grafisch dargestellt. Die hier gemessenen Werte liegen deutlich über den erwarteten Werte.

9.2 Auswertung der Messungen

Die in Abschnitt 9.1 erhobenen Daten sollen nun ausgewertet werden. Die ersten drei Testszenarien messen die Zeit, die ein Sensorknoten benötigt um zu erkennen, ob ein anderer Knoten erreichbar ist oder nicht. Die hier gemessenen Werte entsprechen den erwarteten Werten. Die Testszenarien 4 und 5 sollen überprüfen, ob der aufspannende Baum in der erwarteten Zeit aufgebaut wird. Bei diesen Tests gibt es deutliche Abweichungen von den erwarteten Messwerten. Auch bei Testszenarien 6 und 7, die Messungen bezüglich des Transformations- und Nutzalgorithmus erheben, konnten deutliche Abweichungen festgestellt werden.
Genauere Untersuchungen ergaben, dass die teilweise starken Abweichungen durch Probleme bei der Übertragung von Funknachrichten ihren Ursprung haben. Viele der zwischen den Sensorknoten gesendeten Nachrichten kommen nicht bei ihrem Empfänger an. Diese Probleme traten nicht in der Simulation des Algorithmus mit dem Simulationsprogramm Cooja auf. Das legt nahe, dass die Probleme mit der Hardware der Sensorknoten zusammenhängt.

Obwohl Nachrichten verloren gehen, stabilisierte sich das System in allen untersuchten Testszenarien. Der Algorithmus ist in der Lage den Verlust der Nachrichten zu kompensieren. Da die Fehler in größeren Netzwerken und mit steigender Höhe des aufspannenden Baums zunehmen, wird die Stabilisierung des Systems immer länger dauern. Es ist wahrscheinlich, dass das System ab einer bestimmten Größe des Sensornetzwerkes, aufgrund verlorener Nachrichten nicht mehr stabilisieren kann.

Testszenarien mit mehr als fünf Sensorknoten und mehr als drei Ebenen des aufspannenden Baumes konnten in dem Rahmen dieser Arbeit nicht untersucht werden.

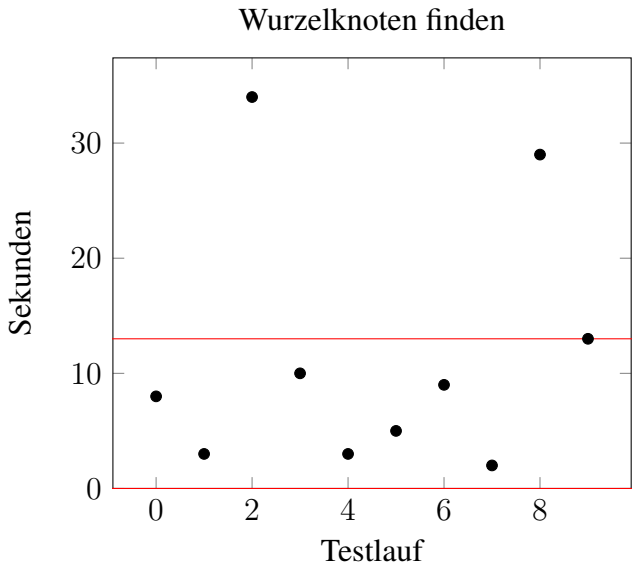


Abbildung 9.6: Transformationsalgorithmus 1

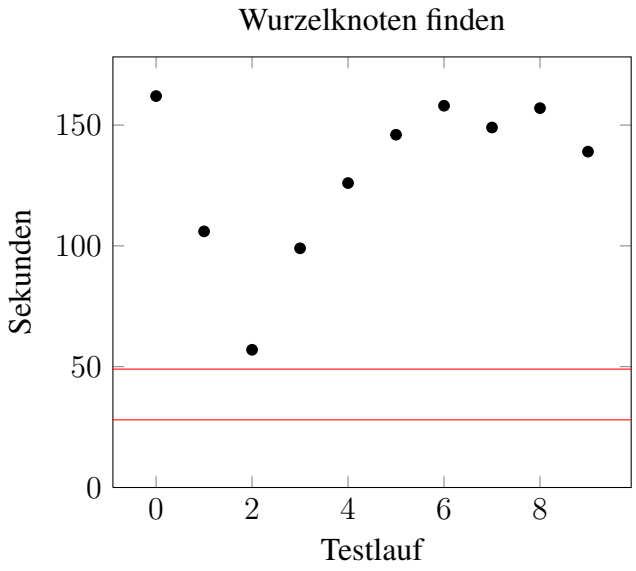


Abbildung 9.7: Transformationsalgorithmus 2

10 Zusammenfassung und Ausblick

Ein selbststabilisierendes Sensornetzwerk ist ein fehlertolerantes Netzwerk aus Sensorknoten. Ein solches Netzwerk ist in der Lage sich nach dem Auftreten von Fehlern selbst wieder in einen funktionsfähigen Zustand zu versetzen. Durch ihre Fähigkeit sich von vorübergehend auftretenden Fehlern zu „erholen“, können selbststabilisierende Sensornetzwerke über Monate hinweg autonom arbeiten.

In dieser Arbeit wurde ein selbststabilisierender Algorithmus für ein Sensornetzwerk entwickelt, der zu Demonstrationszwecken des selbststabilisierenden Mechanismus dient.

Selbststabilisierende Algorithmen benötigen oftmals ein bestimmtes Umfeld, in dem sie funktionieren. Viele dieser Algorithmen setzen das Vorhandensein eines zentralen Steuerprogramms (engl. Scheduler) und den Informationsaustausch mithilfe von geteiltem Speicher (engl. shared memory) voraus. Ein Sensornetzwerk besitzt kein Steuerprogramm und tauscht Informationen mithilfe von Nachrichten aus (engl. messagepassing). Der in dieser Arbeit entwickelte Algorithmus kann in drei Teilalgorithmen unterteilt werden: dem Routing-Algorithmus, dem Transformationsalgorithmus und dem Nutzalgorithmus. Während der erste Algorithmus dafür zuständig ist, dass alle Knoten untereinander kommunizieren können, simulieren der zweite für den Nutzalgorithmus ein Umfeld mit einem zentralen Steuerprogramm und geteiltem Speicher. Durch diesen Aufbau ist es mit wenigen Anpassungen möglich den Nutzalgorithmus gegen einen mit den selben Anforderungen auszutauschen.
Der Nutzalgorithmus berechnet auf jedem Knoten den Durchschnittswert der Helligkeitssensoren aller Knoten. Das Ergebnis wird auf ein an den Sensorknoten angebrachtes Display angezeigt. Die Sensorknoten sind in einer Schutzhülle verbaut, an der ein Taster angebracht ist. Durch Drücken dieses Tasters kann ein neuer Helligkeitswert an dem Sensorknoten eingestellt werden, sodass sich das System wieder auf diesen neuen Wert stabilisieren muss. Einem Beobachter ist es dadurch möglich, den selbststabilisierenden Mechanismus zu beobachten.

Um die Selbststabilisierung des Sensornetzwerkes zu überprüfen wurden Testszenarien, in denen Messungen durchgeführt werden, entwickelt. Beim

Vergleich von erwarteten Werten und gemessenen Werten wurden in einigen Testszenarien starke Abweichungen festgestellt. Diese Abweichungen konnten auf Probleme bei der Funkübertragung von Nachrichten zurückgeführt werden. Trotz dieser Probleme konnte der Algorithmus sich in allen Testszenarien stabilisieren. Dieses Ergebnis verdeutlicht, dass ein selbststabilisierender Algorithmus, trotz transienter Fehler, wie z.B. Übertragungsfehler, weiterhin funktionsfähig bleibt, was wichtig für viele autonome Systeme ist.

Ausblick Die im Rahmen dieser Arbeit festgestellten Probleme bei der Funkübertragung wirken sich stark auf die Zeit aus, die der Algorithmus zum Stabilisieren braucht. Eine weiterführende Abschlussarbeit könnte diese Probleme beheben. Denkbar wäre ein Austausch des verwendeten Rime-Protokolls mit einem anderen Netzwerkprotokoll (z.B. uIP). Da auch einige Probleme mit dem Betriebssystem Contiki-OS in Zusammenhang mit den MTM-CM5000MSP Sensorknoten festgestellt wurden, wäre auch eine Portierung des Algorithmus auf das TinyOS Betriebssystem denkbar.

Literaturverzeichnis

[AB97] AFEK, Yehuda ; BREMLER, Anat: Self-stabilizing unidirectional network algorithms by power-supply. In: *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics (SODA '97). – ISBN 0–89871–390–0, 111–120

[AV91] AWERBUCH, B. ; VARGHESE, G.: Distributed programchecking: a paradigm for building self-stabiIizing distributed protocols. In: *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, 1991, S. 258–267

[BAV91] B. AWERBUCH, B. Patt-Shamir ; VARGHESE, G.: Selfstabilization by local checking and correction. In: *Proc. of the 32nd IEEE Ann. Symp. on Foundation of Computer Science*, 1991, S. 268–277

[Con] *Contiki-OS*. `http://www.contiki-os.org/`. – Zuletzt besucht am 25.08.2013

[DFR] DFROBOT: *Liquid Crystal I2C Library*. `http://www.dfrobot.com`. – Zuletzt besucht am 23.09.2013

[Dij74] DIJKSTRA, Edsger W.: Self-stabilizing systems in spite of distributed control. In: *Commun. ACM* 17 (1974), November, Nr. 11, 643–644. `http://dx.doi.org/10.1145/361179.361202`. – DOI 10.1145/361179.361202. – ISSN 0001–0782

[expa] *I2C LCD 1602 Module , Exp-Teck*. `http://www.exp-tech.de/Displays/I2C-LCD-1602-Module.html`. – Zuletzt besucht am 29.08.2013

[expb] *Pololu 5V Step Up Step Down Spannungsregler, Exp-Teck*. `http://www.exp-tech.de/`. – Zuletzt besucht am 12.09.2013

[inc] INC., Maxford T.: *MTM-CM5000-MSP*. `http://maxfor.co.kr/eng/en_sub5_1_1.html`. – Zuletzt besucht am 19.08.2013

[MK96] MIZUNO, Masaaki ; KAKUGAWA, Hirotsugu: A timestamp based transformation of self-stabilizing programs for distributed computing environments. Version: 1996. `http://dx.doi.org/10.1007/3-540-61769-8_20`. In: BABAOĞLU, Özalp (Hrsg.) ; MARZULLO, Keith (Hrsg.): *Distributed Algorithms* Bd. 1151. Springer Berlin Heidelberg, 304-321. – DOI 10.1007/3–540–61769–8_20. – ISBN 978–3–540–61769–3

[Sch93] SCHNEIDER, Marco: Self-stabilization. In: *ACM Comput. Surv.* 25 (1993), März, Nr. 1, 45–67. `http://dx.doi.org/10.1145/151254.151256`. – DOI 10.1145/151254.151256. – ISSN 0360–0300

[Sem] SEMICONDUCTORS, NXP: *I2C-bus specification and user manual*. http://www.nxp.com/documents/other/UM10204_v5.pdf. – Zuletzt besucht am 25.09.2013

[Tin] *TinyOS*. http://www.tinyos.net/. – Zuletzt besucht am 25.08.2013

[Wu08] WU, Randy: *MSP430 Interface to TPS60250 via I2C Master Software*. http://www.ti.com/lit/an/slva302/slva302.pdf. Version: 2008

A Ergebnisse der Messungen

A.1 Timeout-Werte

Timeout-Prozess	Timeout in Sekunden
Broadcast-Timeout	20 - 36
Spanningtree-Timeout	15 - 20
Routing-Timeout	25 - 30
Messagepassing-Timeout	8 - 13

Tabelle A.1: Timeout-Werte

A.2 Messergebnisse

	Knoten A	Knoten B	
Nr.	BC-Nachricht	BC-Timeouts	Zeit in s
1	1	1	27
2	1	1	34
3	1	1	21
4	1	1	22
5	1	1	29

Tabelle A.2: Knotenfindung mit zwei Knoten

	Knoten A	Knoten B	Knoten C	Knoten D	Knoten E
Nr.	BC-Nachrichten	BC-Timeouts	BC-Timeouts	BC-Timeouts	BC-Timeouts
1	4	1	1	1	1
2	4	1	1	1	1
3	4	1	1	1	1
4	4	1	1	1	1
5	4	1	1	1	1

Tabelle A.3: Knotenfindung mit fünf Knoten

	Knoten A
Nr.	BC-Timeouts
1	3
2	3
3	3
4	3
5	3

Tabelle A.4: Knotenlöschung mit zwei Knoten

	Knoten A
Nr.	BC-Timeouts
1	3
2	3
3	3
4	3
5	3

Tabelle A.5: Knotenlöschung mit fünf Knoten

	Knoten A		Knoten B		
Nr.	ST-Timeouts	ST-Nachrichten	ST-Timeouts	ST-Nachrichten	Zeit in s
1	6	10	7	3	107
2	7	9	6	3	117
3	10	7	10	3	156
4	5	7	4	3	80
5	5	8	5	3	81
6	6	8	6	4	100
7	5	8	5	3	85
8	7	6	6	3	112
9	5	5	5	3	82
10	5	7	5	3	76

Tabelle A.6: Wurzelknoten finden mit zwei Knoten

	Knoten A		Knoten B		
Nr.	ST-Timeouts	ST-Nachrichten	ST-Timeouts	ST-Nachrichten	Zeit in s
1	27	19	22	7	383
2	16	14	11	6	189
3	17	21	14	6	230
4	31	11	27	9	467
5	42	31	37	14	640

Tabelle A.7: Wurzelknoten finden mit drei Knoten

	Knoten A	
Nr.	MP-Timeouts	Zeit in s
1	1	8
2	1	3
3	4	34
4	1	10
5	1	3
6	1	5
7	1	9
8	1	2
9	3	29
10	1	13

Tabelle A.8: Messagepassing mit zwei Knoten

	Knoten A
Nr.	Zeit in s
1	162
2	106
3	57
4	99
5	126
6	146
7	158
8	149
9	157
10	139

Tabelle A.9: Messagepassing mit fünf Knoten

B Bedienungsanleitung

In dieser kleinen Bedienungsanleitung wird erklärt, wie der Algorithmus auf die Sensorknoten übertragen wird.

B.1 Den Algorithmus auf das Sensornetzwerk übertragen

B.1.1 Voraussetzungen

Um den Algorithmus auf das Sensornetzwerk übertragen zu können, muss dieser zunächst kompiliert werden. Dazu wird ein Computer mit einem Linux System benötigt, auf dem Contiki-OS installiert ist. Genauere Anweisungen, wie Contiki-OS eingerichtet wird, sind auf der Webseite [Con] zu finden. Alternative kann Instant-Contiki verwendet werden. Instant-Contiki ist eine VM (virtualle Maschine) mit einem Ubuntu als Betriebssystem. Auf der virtuellen Maschine ist alle für Contiki-OS benötigten Software installiert und eingerichtet. Eine Kopie der in dieser Arbeit verwendeten VM befindet sich auf der dieser Arbeit beiliegenden DVD.

B.1.2 Kompilierung des Algorithmus

Ist Contiki-OS eingerichtet, öffnet man ein Terminal und wechselt in das Verzeichnis mit dem Quelltext. In der VM liegt der Quelltext in dem Verzeichnis „/home/user/contikiprojects/Selbststabilisierendes Sensornetzwerk/“. Wird die VM nicht benutzt, muss ggf. im Makefile der Pfad zu dem Contiki-Verzeichnis noch angepasst werden. Der Quelltext kann mit dem Befehl

```
make node.upload
```

kompiliert werden. In der VM werden zusätzliche Berechtigungen für den Zugriff auf die USB-Schnittstelle benötigt, darum muss der Befehl

```
sudo make node.upload
```

lauten. Dieser Befehl kompiliert den Quelltext und schreibt ihn danach auf die per USB angeschlossenen Sensorknoten.

B.2 Vom Computer aus mit den Sensorknoten kommunizieren

Ist ein Sensorknoten mit dem Computer per USB-Schnittelle verbunden, so ist es möglich eine serielle Verbindung zu diesem aufzubauen. Um sich mit dem Sensorknoten zu verbinden öffnet man ein Terminal und geht in den Verzeichnis des Quelltexts. In diesem Verzeichnis muss der Befehl

```
sudo make login node.upload
```

eingegeben werden. Sind mehrere Sensorknoten angeschlossen, kann mit dem Parameter „MOTES“ der USB-Anschluss ausgewählt werden.

```
sudo make login node.upload MOTES=/dev/ttyUSB0
```

Nachdem die Verbindung aufgebaut ist, können die Befehle „ping“ und „info“ verwendet werden.

B.3 Einen neuen Sensorknoten im Sensornetzwerk aufnehmen

Soll ein neuer Sensorknoten dem Sensornetzwerk hinzugefügt werden, so ist eine Anpassung der Lese-List nötig. Der neue Sensorknoten muss den anderen Sensorknoten bekannt gemacht werden. Dazu muss zuerst die Adresse des Sensorknotens herausgefunden werden. Dazu wird, wie in Abschnitt B.2 beschrieben, eine Verbindung zu dem Sensorknoten aufgebaut. Ist diese Verbindung aufgebaut wird der Knoten durch Drücken des roten Tasters neu gestartet. Nach dem Neustart gibt der Knoten einige Informationen über die serielle Schnittstelle aus, darunter auch seine Adresse.

Diese muss nun in der Datei „node.c“ im Prozess „MP Timeout“ Prozess nachgetragen werden. Dort wird eine neue Bedingung eingefügt.

```
\item if (rimeaddr_node_addr.u8[0] == x1 &&
   rimeaddr_node_addr.u8[1] == x2) {
   add_reader(y1,y2);
}
```

Die beiden mit $x1$ und $x2$ gekennzeichneten Stellen sind mit den beiden Teilen der Adresse zu ersetzen. Mit der Funktion „add_reader“ können Knoten zur Lese-Liste hinzugefügt werden. Dem neuen Knoten müssen alle anderen Knoten mit diesem Befehl hinzugefügt werden. Außerdem muss der neue Knoten zu allen anderen Knoten hinzugefügt werden.

C Quelltext des selbststabilisierenden Algorithmus

Listing C.1: node.h

```
#include "contiki.h"
#include "lib/list.h"
#include "lib/memb.h"
#include "lib/random.h"
#include "net/rime.h"

#include <stdio.h>

#include "dev/button-sensor.h"
#include "dev/leds.h"
#include "lib/random.h"

#include "dev/serial-line.h"

#define MAX_NEIGHBORS 16


/*******      Definition der Prozesse      ********/
PROCESS(ST_timeout_process, "ST Timeout");
PROCESS(RP_timeout_process, "RP Timeout");
PROCESS(serial_command_process, "Serial Commands");
PROCESS(send_process, "Send Process");
PROCESS(BT_timeout_process, "BC Process");
PROCESS(MP_timeout_process, "MP Timer");
PROCESS(button_process, "Button");

/*******      Prozesse starten      ********/
AUTOSTART_PROCESSES(&ST_timeout_process, &
   RP_timeout_process, &serial_command_process, &
   send_process, &BT_timeout_process, &
   MP_timeout_process, &button_process);



static process_event_t event_senddata_ready;

static struct broadcast_conn broadcast;
static struct runicast_conn runicast;


/*******        Strukturen und Funktionen
   *********/
struct neighbor {
```

```
  struct neighbor *next;    //Wird fuer Contiki-Liste
     benoetigt
  rimeaddr_t addr;
  rimeaddr_t routing_port;
  uint16_t timeouts;
};



/***** Variablen ST ********/
static rimeaddr_t parent;
static rimeaddr_t current_leader;
static rimeaddr_t prev;
static uint8_t cl_dist;
static uint8_t prev_dist;

static rimeaddr_t tmp_parent;

MEMB(neighbors_memb, struct neighbor, MAX_NEIGHBORS);

LIST(neighbors_list);

MEMB(prev_ports_memb, struct neighbor, MAX_NEIGHBORS);

LIST(prev_ports_list);

MEMB(routing_memb, struct neighbor, MAX_NEIGHBORS);

LIST(routing_list);

/****** Broadcast-Nachricht *******/
struct message {
  struct message *next;
  uint8_t seq;
  rimeaddr_t from;
  rimeaddr_t to;
  rimeaddr_t port;
  rimeaddr_t current_leader;
  uint8_t cl_dist;
  uint8_t type;
  uint16_t ts;
  uint16_t q;
};

enum {
  ST_WEAK,  //0
  ST_STRONG,    //1 Spanning Tree
  BC,    //2
  RP_UPDATE,    //3 Routing-Protokoll
  SYS_PING, //4 System-Nachricht
  SYS_PONG, //5 System-Nachricht
  MP_GRANT, //6
  MP_ABORT, //7
```

```
  MP_COMMIT,   //8
  MP_STATE, //9
  MP_REQ_COMMIT,  //10
};
```

Listing C.2: node.c

```
/*
selbststabilisierender Algorithmus
von Marius Hacker
*/
#define DEBUG 1
#include "node.h"
#include "dev/leds.h"

#include "functions.h"
#include "spanningtree.h"
#include "routing.h"
#include "messagepassing.h"

#include "MSP430_SWI2C_Master.h"
#include "Lcd.h"
#include "dev/light-sensor.h"

#define MYDEBUGLEVEL 0


static struct message msg;


/*******      Broadcast-Callback-Funktion      *******
   */
/* Diese Funktion wird aufgerufen sobald eine Runicast-
   Nachricht eintrifft */
static void runicast_recv(struct broadcast_conn *c,
   const rimeaddr_t *from)
{
   struct message *remsg;

   remsg = packetbuf_dataptr();

   msg.next = NULL;
   msg.from = remsg->from;
   msg.port = *from;
   msg.to = remsg->to;
   msg.q = remsg->q;
   msg.ts = remsg->ts;
   rimeaddr_copy(&msg.current_leader, &remsg->
      current_leader);
   msg.cl_dist = remsg->cl_dist;
   msg.type = remsg->type;


   #if (MYDEBUGLEVEL >= 1)
```

```
   printf("msg: from: %d.%d to: %d.%d port: %d.%d
      leader: %d.%d -- type: %d -- dist: %d\n", msg.
      from.u8[0], msg.from.u8[1], msg.to.u8[0], msg.to.
      u8[1], msg.port.u8[0], msg.port.u8[1], msg.
      current_leader.u8[0], msg.current_leader.u8[1],
      msg.type, msg.cl_dist);
   #endif

   if (!rimeaddr_cmp(&msg.to, &rimeaddr_node_addr)) {
   //Wenn Nachricht nicht an Sensorknoten, dann
      Nachricht routen.
      send(msg);
50 } else if (msg.type == ST_WEAK || msg.type ==
      ST_STRONG) {
   //Spanningtree-Nachricht verarbeiten
      spanningtree(msg);

   } else if (msg.type == SYS_PING) {
   //Ping-Nachricht verarbeiten
      msg.type = SYS_PONG;
      rimeaddr_copy(&msg.to, &msg.from);
      rimeaddr_copy(&msg.from, &rimeaddr_node_addr);
      rimeaddr_copy(&msg.port, routing(&msg.to));
60    send(msg);
   } else if (msg.type == SYS_PONG) {
   //Pong-Nachricht verarbeiten
      printf("received PONG from %d.%d\n", msg.from.u8
         [0], msg.from.u8[1]);
   } else if (msg.type == RP_UPDATE) {
   //Routing-Update-Nachricht verarbeiten
         //Routingtabelle aktualisieren
         struct neighbor *nn;
         for(nn = list_head(routing_list); nn != NULL;
            nn = list_item_next(nn)) {
               if(rimeaddr_cmp(&nn->addr, &msg.from)) {
70                   break;
               }
         }
         if (nn == NULL) {
            nn = list_new(routing_list, &routing_memb);
         }
         if (nn == NULL) {
            printf("Kein Spiecher mehr in routing_memb
               !!! groesse: %d\n", list_length(
               neighbors_list));
         }
         rimeaddr_copy(&nn->addr, &msg.from);
80       rimeaddr_copy(&nn->routing_port, &msg.port);
         nn->timeouts = 0;
         if(!rimeaddr_nil(&parent)) {
            rimeaddr_copy(&msg.port, &parent);
            rimeaddr_copy(&msg.to, &parent);
            send(msg);
```

```
            }


   } else if (msg.type == MP_GRANT || msg.type ==
      MP_ABORT || msg.type == MP_COMMIT || msg.type ==
      MP_STATE || msg.type == MP_REQ_COMMIT) {
90 //Messagepassing-Nachricht verarbeiten
      messagepassing(msg);

   }

   if (rimeaddr_cmp(&parent, &tmp_parent) == 0) {
   //Wenn sich der Vaterknoten geaendert hat, neuen
      Wert auf dem Display ausgeben
      lcdprint_parent(&parent);
      rimeaddr_copy(&tmp_parent, &parent);
   }
100



}


/*******      Broadcast-Callback-Funktion      *******
   */
/* Diese Funktion wird aufgerufen sobald eine Broadcast
   -Nachricht eintrifft */
static void broadcast_recv(struct broadcast_conn *c,
   const rimeaddr_t *from)
{
110 struct message *remsg;

   remsg = packetbuf_dataptr();

   //Nachbarliste pflegen
   struct neighbor *nn;
   for(nn = list_head(neighbors_list); nn != NULL; nn =
       list_item_next(nn)) {
      if(rimeaddr_cmp(&nn->addr, from)) {
               break;
         }
120 }
   if (nn == NULL) {
      nn = list_new(neighbors_list, &neighbors_memb);
   }
   if (nn == NULL) {
      printf("Kein Spiecher mehr in neighbor_memb!!!
         groesse: %d\n", list_length(neighbors_list));
   }
   nn->addr = *from;
   nn->timeouts = 0;

130 }
```

```
/*******      Callback-Funktion-Registrieren
   ********/
static const struct broadcast_callbacks broadcast_call
   = {broadcast_recv};
static const struct runicast_callbacks
   runicast_callbacks = {runicast_recv};


/*******      Spanningtree-Timeoutprozess      *******
   */
PROCESS_THREAD(ST_timeout_process, ev, data)
{
  static struct etimer et;

  PROCESS_BEGIN();

  parent.u8[0] = 0;
  parent.u8[1] = 0;
  current_leader.u8[0] = 0;
  current_leader.u8[1] = 0;
  prev.u8[0] = 0;
  prev.u8[1] = 0;
  cl_dist = 0;
  prev_dist = 0;
  list_init(neighbors_list);
  memb_init(&neighbors_memb);
  list_init(prev_ports_list);
  memb_init(&prev_ports_memb);

  while(1) {

   #if (MYDEBUGLEVEL >= 1)
   printf("ST_Timeout\n");
   #endif

    //Timer setzen
    etimer_set(&et, CLOCK_SECOND * 15 + random_rand() %
        (CLOCK_SECOND * 5));

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

    //Starke und schwache Spanningtree-Nachrichten
       senden
    if(parent.u8[0] == 0 && parent.u8[1] == 0) {
   leds_off(LEDS_GREEN);
   send_neighbors(ST_STRONG);
    } else {
   leds_on(LEDS_GREEN);
   send_neighbors(ST_WEAK);
    }
  }
```

```
  PROCESS_END();
}


/*******      Routing-Timeoutprozess      ********/
PROCESS_THREAD(RP_timeout_process, ev, data)
{
  static struct etimer et2;
  static struct message rp_msg;

  PROCESS_BEGIN();

  list_init(routing_list);
  memb_init(&routing_memb);

  while(1) {

   #if (MYDEBUGLEVEL >= 1)
   printf("RP_Timeout\n");
   #endif

    etimer_set(&et2, CLOCK_SECOND * 25 + random_rand()
       % (CLOCK_SECOND * 5));

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et2));


    struct neighbor *n5;
    struct neighbor *n4;
    n5 = list_head(routing_list);
    while (n5 != NULL) {
   n5->timeouts++;
   n4 = n5;
   n5 = list_item_next(n5);
   if (n4->timeouts > 4) {
      list_remove(routing_list, n4);
   }
    }

      if(!rimeaddr_nil(&parent)) {
      printf("routing sende update\n");
      rp_msg.type = RP_UPDATE;
      rimeaddr_copy(&rp_msg.from, &rimeaddr_node_addr);
      rimeaddr_copy(&rp_msg.to, &parent);
      rimeaddr_copy(&rp_msg.port, &parent);
      send(rp_msg);
      }
  }

  PROCESS_END();
}
```

```
230
/*******      Broadcast-Timeoutprozess      ********/
PROCESS_THREAD(BT_timeout_process, ev, data)
{
  static struct etimer et;
  struct message msg;

  PROCESS_EXITHANDLER(broadcast_close(&broadcast);)

240  PROCESS_BEGIN();

  broadcast_open(&broadcast, 129, &broadcast_call);


  while(1) {

    etimer_set(&et, CLOCK_SECOND * 20 + random_rand() %
        (CLOCK_SECOND * 16));

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
250
    struct neighbor *n5;
    struct neighbor *n4;
    n5 = list_head(neighbors_list);
    while (n5 != NULL) {
   n5->timeouts++;
   n4 = n5;
   n5 = list_item_next(n5);
   if (n4->timeouts > 3)
      list_remove(neighbors_list, n4);
260
    }

    packetbuf_copyfrom(&msg, sizeof(struct message));
    msg.type = BC;
    broadcast_send(&broadcast);
  }

  PROCESS_END();
}
270

/*******      Serialler-Prozess      ********/
/* Nimmt eingaben ueber die serielle Schnitstelle
   entgegen */
PROCESS_THREAD(serial_command_process, ev, data)
 {
   static struct message mymsg;

   PROCESS_BEGIN();

280   leds_init();
```

```
   LiquidCrystal_I2C(0x27, 8, 2);
   init();
   noCursor();
   noBlink();
   backlight();

//Begruessungsausgabe auf dem Display
  lcdprint("Sensornetzwerk", 0);
  lcdprint("by Marius Hacker", 1);

   static struct etimer lcdet;

    etimer_set(&lcdet, CLOCK_SECOND * 2);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&lcdet));

    lcdprint("initialisiere", 0);
    lcdprint("Schaltkreise", 1);

    etimer_set(&lcdet, CLOCK_SECOND * 2);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&lcdet));

    lcdprint("kalibriere", 0);
    lcdprint("Sensoren", 1);

     //Helligkeitswert auslesen
     SENSORS_ACTIVATE(light_sensor);
     clock_delay(20000);
     q = light_sensor.value(0);
     SENSORS_DEACTIVATE(light_sensor);

    etimer_set(&lcdet, CLOCK_SECOND * 2);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&lcdet));

    lcdprint("starte", 0);
    lcdprint("Fluxkompensator", 1);

 etimer_set(&lcdet, CLOCK_SECOND * 2);

    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&lcdet));

  clear();
  lcdprint_addr(&rimeaddr_node_addr);
  lcdprint_parent(&parent);
  lcdprint_sensorw(q);
  lcdprint_durchschn(average());

 //Warte auf Eingaben und behandel diese
 for(;;) {
    PROCESS_YIELD();
```

```
    if(ev == serial_line_event_message) {
  char delimiter[] = " ";
  char *ptr;

  ptr = strtok((char *)data, delimiter);

  if (strncmp(ptr, "ping", 4) == 0) {
     ptr = strtok(NULL, delimiter);
     if (ptr != NULL) {

        delimiter[0] = '.';
        int rimeadr0 = atoi(strtok(ptr, delimiter));

        int rimeadr1 = atoi(strtok(NULL, delimiter));

        printf("Sende ping zu %d.%d\n", rimeadr0,
           rimeadr1);
        while (runicast_is_transmitting(&runicast))
           PROCESS_PAUSE();

        mymsg.type = SYS_PING;
        mymsg.to.u8[0] = rimeadr0;
        mymsg.to.u8[1] = rimeadr1;
        rimeaddr_copy(&mymsg.from, &rimeaddr_node_addr
           );
        rimeaddr_copy(&mymsg.port, routing(&mymsg.to))
           ;
        send(mymsg);
     }
  } else if (strncmp(ptr, "info", 4) == 0) {
     printf("Parent: %d.%d, Leader: %d.%d\n", parent.
        u8[0], parent.u8[1], current_leader.u8[0],
        current_leader.u8[1]);
  }

    }
  }
  PROCESS_END();
 }

/*******      Sende-Prozess      ********/
/* Zustaendig fuer das Senden von Nachrichten */
PROCESS_THREAD(send_process, ev, data)
 {
  static struct message send_msg;

 PROCESS_EXITHANDLER(runicast_close(&runicast);)

 PROCESS_BEGIN();
leds_init();

  event_senddata_ready = process_alloc_event();
```

```
runicast_open(&runicast, 144, &runicast_callbacks);

 while (1) {
    PROCESS_WAIT_EVENT_UNTIL(ev ==
       event_senddata_ready || ev ==
       PROCESS_EVENT_POLL);

    send_msg = *(struct message*)data;

    if (send_msg.type == ST_STRONG || send_msg.type
       == ST_WEAK) {
       static struct neighbor *n6;

       for(n6 = list_head(neighbors_list); n6 != NULL
          ; n6 = list_item_next(n6)) {
          while (runicast_is_transmitting(&runicast))
                PROCESS_PAUSE();
          rimeaddr_copy(&send_msg.to , &n6->addr);
          packetbuf_copyfrom(&send_msg, sizeof(struct
              message));

          runicast_send(&runicast, &n6->addr, 1);
       }
    } else if (send_msg.type == MP_REQ_COMMIT &&
       rimeaddr_cmp(&send_msg.from, &
       rimeaddr_node_addr)) {
       static struct mp_neighbor *m6;

       for(m6 = list_head(writers_list); m6 != NULL;
          m6 = list_item_next(m6)) {
          while (runicast_is_transmitting(&runicast))
                PROCESS_PAUSE();
          rimeaddr_copy(&send_msg.to , &m6->addr);
          rimeaddr_copy(&send_msg.port, routing(&
             send_msg.to));
          if (!rimeaddr_nil(&send_msg.port)) {
             packetbuf_copyfrom(&send_msg, sizeof(
                struct message));

             runicast_send(&runicast, routing(&
                send_msg.to), 1);
          }
       }
    } else if ((send_msg.type == MP_COMMIT ||
       send_msg.type == MP_STATE) && rimeaddr_cmp(&
       send_msg.from, &rimeaddr_node_addr)) {
       static struct mp_neighbor *m7;

       for(m7 = list_head(readers_list); m7 != NULL;
          m7 = list_item_next(m7)) {
          while (runicast_is_transmitting(&runicast))
                PROCESS_PAUSE();
          rimeaddr_copy(&send_msg.to , &m7->addr);
```

```
            rimeaddr_copy(&send_msg.port, routing(&
               send_msg.to));
            if (!rimeaddr_nil(&send_msg.port)) {
               packetbuf_copyfrom(&send_msg, sizeof(
                  struct message));
               runicast_send(&runicast, routing(&
                  send_msg.to), 1);
            }
         }
      } else  {
         while (runicast_is_transmitting(&runicast))
                  PROCESS_PAUSE();
         packetbuf_copyfrom(&send_msg, sizeof(struct
            message));
         if (!rimeaddr_nil(routing(&send_msg.to))) {
            runicast_send(&runicast, routing(&send_msg.
               to), 1);
         }
      }
   }
   PROCESS_END();
 }


/*******      Messagepassing-Timeoutprozess
   ********/
PROCESS_THREAD(MP_timeout_process, ev, data)
{

  static struct message rp_msg2;


  PROCESS_BEGIN();


  list_init(writers_list);
  memb_init(&writers_memb);
   list_init(readers_list);
  memb_init(&readers_memb);

  etimer_set(&et3, CLOCK_SECOND * 8 + random_rand() % (
     CLOCK_SECOND * 5));
  etimer_set(&et4, CLOCK_SECOND * 20 + random_rand() %
     (CLOCK_SECOND * 5));
  etimer_set(&nutzalgo, CLOCK_SECOND * 30 + random_rand
     () % (CLOCK_SECOND * 5));


/* Eintraege in der Schreib-, Lese-Liste */
if (rimeaddr_node_addr.u8[0] == 211 &&
   rimeaddr_node_addr.u8[1] == 192) {
   add_reader(32,71);
   add_reader(74,72);
```

```
   add_reader(207,40);
   add_reader(249,88);
  }
  if (rimeaddr_node_addr.u8[0] == 32 &&
     rimeaddr_node_addr.u8[1] == 71) {
   add_reader(74,72);
   add_reader(211,192);
   add_reader(207,40);
   add_reader(249,88);
  }
if (rimeaddr_node_addr.u8[0] == 207 &&
   rimeaddr_node_addr.u8[1] == 40) {
   add_reader(32,71);
   add_reader(74,72);
   add_reader(211,192);
   add_reader(249,88);
  }

  if (rimeaddr_node_addr.u8[0] == 74 &&
     rimeaddr_node_addr.u8[1] == 72) {
   add_reader(32,71);
   add_reader(211,192);
   add_reader(207,40);
   add_reader(249,88);
  }
if (rimeaddr_node_addr.u8[0] == 249 &&
   rimeaddr_node_addr.u8[1] == 88) {
   add_reader(32,71);
   add_reader(74,72);
   add_reader(211,192);
   add_reader(207,40);
  }



  while(1) {

   #if (MYDEBUGLEVEL >= 1)
   printf("MP Timeout \n");
   #endif

    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER)
       ;
   if (etimer_expired(&et3)) {

         rp_msg2.q = q;
      rp_msg2.ts = req_ts;
      rp_msg2.type = MP_STATE;
      rimeaddr_copy(&rp_msg2.from, &rimeaddr_node_addr)
         ;
      send(rp_msg2); //COMMIT
```

```
      etimer_set(&et3, CLOCK_SECOND * 8 + random_rand()
          % (CLOCK_SECOND * 5));
   } else if (etimer_expired(&et4)) {

      mode = NO_PRIV;
      etimer_set(&et4, CLOCK_SECOND * 20 + random_rand
         () % (CLOCK_SECOND * 5));
   }
  }

  PROCESS_END();
}



/*******      Button-Prozess      ********/
/* Zustaendig fuer das Abfragen des Tasters */
static uint8_t active;
PROCESS_THREAD(button_process, ev, data)
{
  PROCESS_BEGIN();
  active = 0;
  SENSORS_ACTIVATE(button_sensor);


  while(1) {
    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
       data == &button_sensor);
     SENSORS_ACTIVATE(light_sensor);
     clock_delay(20000);
     q = light_sensor.value(0);
     lcdprint_sensorw(q);
     SENSORS_DEACTIVATE(light_sensor);
     lcdprint_durchschn(average());

  }
  PROCESS_END();
}
```

Listing C.3: functions.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

//Rime-Adresse auf null pruefen
int rimeaddr_nil(rimeaddr_t *a) {
   if (a->u8[0] == 0 && a->u8[1] == 0)
      return 1;
   return 0;
}

//Pruefen ob Rime-Adresse-A lexikografisch kleiner oder
    gleich Rime-Adresse-B
```

```
int rimeaddr_lexi_le(rimeaddr_t *a, rimeaddr_t *b,
   uint8_t dista, uint8_t distb) {
   if (a->u8[0] < b->u8[0])
      return 1;
   else if (a->u8[0] == b->u8[0]) {
      if (a->u8[1] < b->u8[1])
         return 1;
      if (a->u8[1] == b->u8[1])
         if (dista <= distb)
            return 1;
   }
   return 0;
}

//Pruefen ob Rime-Adresse-A lexikografisch kleiner Rime
   -Adresse-B
int rimeaddr_lexi_l(rimeaddr_t *a, rimeaddr_t *b,
   uint8_t dista, uint8_t distb) {
   if (a->u8[0] < b->u8[0])
      return 1;
   else if (a->u8[0] == b->u8[0]) {
      if (a->u8[1] < b->u8[1])
         return 1;
      if (a->u8[1] < b->u8[1])
         if (dista <= distb)
            return 1;
   }
   return 0;
}

//Liste loeschen
void list_clear(list_t list, struct memb *m) {
   struct neighbor *n1 = list_head(list);
   while(list_length(list) > 0) {
      list_pop(list);
      memb_free(m, n1);
      n1 = list_head(list);
   }
}

//Neuen Eintrag in Liste anlegen
struct neighbor* list_new(list_t list, struct memb *m)
   {
   struct neighbor *n1;
   n1 = memb_alloc(m);
   list_add(list, n1);
   return n1;
}

//Ueberpruefen, ob Adresse in Nachbarschaftsliste
int ist_alive(rimeaddr_t *a) {
   struct neighbor *n;
```

```
60    for(n = list_head(neighbors_list); n != NULL; n =
         list_item_next(n)) {
            if(rimeaddr_cmp(&n->addr, a)) {
                  return 1;
            }
      }
      return 0;
}


int ist_in_prev(rimeaddr_t *a) {
70    struct neighbor *n;
      for(n = list_head(prev_ports_list); n != NULL; n =
         list_item_next(n)) {
            if(rimeaddr_cmp(&n->addr, a)) {
                  return 1;
            }
      }
      return 0;
}

//Ueberpruefen, ob Adresse in Prev-Liste
80 struct neigbor* ist_in_neighbors(rimeaddr_t *a) {
      struct neighbor *n;
      for(n = list_head(neighbors_list); n != NULL; n =
         list_item_next(n)) {
         printf("a: %d.%d == b: %d.%d\n", a->u8[0], a->u8
            [1], n->addr.u8[0], n->addr.u8[1]);
            if(rimeaddr_cmp(&n->addr, a)) {
                  return &n->addr;
            }
      }
      return NULL;
}
90

//Nachricht an Nachbarn senden
void send_neighbors(uint8_t type) {
      static struct message sendmsg;

      sendmsg.type = type;

      rimeaddr_copy(&sendmsg.current_leader, &
         current_leader);
      sendmsg.cl_dist = cl_dist + 1;
      #if (MYDEBUGLEVEL >= 1)
100   printf("Sende zu Nachbarn %d -- leader: %d.%d --
         dist: %d\n", type, sendmsg.current_leader.u8[0],
             sendmsg.current_leader.u8[1], sendmsg.cl_dist
         );
      #endif

      process_post(&send_process, event_senddata_ready, &
         sendmsg);
```

```
}

//Nachricht an Knoten senden
void send(struct message sendmsg1) {
   static struct message sendmsg2;
   rimeaddr_copy(&sendmsg2.from, &sendmsg1.from);
   rimeaddr_copy(&sendmsg2.to, &sendmsg1.to);
   rimeaddr_copy(&sendmsg2.port, &sendmsg1.port);
   sendmsg2.type = sendmsg1.type;
   sendmsg2.q = sendmsg1.q;
   sendmsg2.ts = sendmsg1.ts;

   process_post_synch(&send_process,
      event_senddata_ready, &sendmsg2);
}

//Adresse auf LCD ausgeben
void lcdprint_addr(rimeaddr_t *a) {
   char lcdbuffer [7];
   sprintf(lcdbuffer, "       ");
   setCursor(0,0);
   lcd_write(lcdbuffer);
   sprintf(lcdbuffer, "%d.%d", a->u8[0], a->u8[1]);
   setCursor(0,0);
   lcd_write(lcdbuffer);
}

//Vaterknoten auf LCD ausgeben
void lcdprint_parent(rimeaddr_t *a) {
   char lcdbuffer [7];
   sprintf(lcdbuffer, "       ");
   setCursor(9,0);
   lcd_write(lcdbuffer);
   sprintf(lcdbuffer, "%d.%d", a->u8[0], a->u8[1]);
   setCursor(9,0);
   lcd_write(lcdbuffer);
}

//Sensorwert auf LCD ausgeben
void lcdprint_sensorw(uint8_t wert) {
   char lcdbuffer [3];
   sprintf(lcdbuffer, "   ");
   setCursor(0,1);
   lcd_write(lcdbuffer);
   sprintf(lcdbuffer, "%d", wert);
   setCursor(0,1);
   lcd_write(lcdbuffer);
}

//Durchschnittswert auf LCD ausgeben
void lcdprint_durchschn(uint8_t wert) {
   char lcdbuffer [3];
   sprintf(lcdbuffer, "   ");
```

```
   setCursor(4,1);
   lcd_write(lcdbuffer);
   sprintf(lcdbuffer, "%d", wert);
   setCursor(4,1);
   lcd_write(lcdbuffer);
}

//Text auf LCD ausgeben
void lcdprint(char *str, uint8_t line) {
   char lcdbuffer [16];
   sprintf(lcdbuffer, "                ");
   setCursor(0,line);
   lcd_write(lcdbuffer);
   sprintf(lcdbuffer, "%s", str);
   setCursor(0,line);
   lcd_write(lcdbuffer);
}



#endif
```

Listing C.4: spanningtree.h

```
#ifndef SPANNINGTREE_H
#define SPANNINGTREE_H

//Spanningtree-Nachricht verarbeiten
void spanningtree(struct message msg) {
if (msg.type == ST_WEAK || msg.type == ST_STRONG) {
      //if parent = nil
      if(parent.u8[0] == 0 && parent.u8[1] == 0) {
         rimeaddr_copy(&current_leader, &
            rimeaddr_node_addr);

      }
      //if [id_v, 0] <=_lexic current_leader
      if (rimeaddr_lexi_le(&rimeaddr_node_addr, &
         current_leader, 0, cl_dist)) {
         rimeaddr_copy(&current_leader, &
            rimeaddr_node_addr);
         parent.u8[0] = 0;
         parent.u8[1] = 0;
         #if (MYMYDEBUGLEVEL >= 2)
            printf("ST: 2\n");
         #endif
      }
      //if (p = parent) and (msg = current_leader)
      if (!rimeaddr_nil(&msg.current_leader))
         if (rimeaddr_cmp(&msg.port, &parent) &&
            rimeaddr_cmp(&msg.current_leader, &
            current_leader)) {
            if (msg.type == ST_STRONG) {
               send_neighbors(ST_STRONG);
```

```
            } else {
                send_neighbors(ST_WEAK);
            }
        }
    //if (p = parent) and (current_leader != msg)
    if (rimeaddr_cmp(&msg.port, &parent) && !
        rimeaddr_cmp(&msg.current_leader, &
        current_leader) &&  !rimeaddr_nil(&msg.
        current_leader)) {
        rimeaddr_copy(&current_leader, &
            rimeaddr_node_addr);
        parent.u8[0] = 0;
        parent.u8[1] = 0;

        send_neighbors(ST_STRONG);

    }
    //if (msg <_lexic current_leader)
    if (rimeaddr_lexi_l(&msg.current_leader, &
        current_leader, msg.cl_dist, cl_dist) && !
        rimeaddr_nil(&msg.current_leader)) {
        if (msg.type == ST_STRONG && rimeaddr_cmp(&msg
            .current_leader, &prev) && ist_in_prev(&msg
            .port))  {
            rimeaddr_copy(&current_leader, &msg.
                current_leader);
            parent = msg.port;

            send_neighbors(ST_STRONG);
        } else {
            rimeaddr_copy(&current_leader, &
                rimeaddr_node_addr);
            parent.u8[0] = 0;
            parent.u8[1] = 0;

            send_neighbors(ST_STRONG);
        }
    }
    //if (msg <_lexic prev)
    if (rimeaddr_lexi_l(&msg.current_leader, &prev,
        msg.cl_dist, prev_dist) && !rimeaddr_nil(&prev
        ) && !rimeaddr_nil(&msg.current_leader)) {
        if (msg.type == ST_STRONG) {
            rimeaddr_copy(&prev, &msg.current_leader);
            prev_dist = msg.cl_dist;

            list_clear(prev_ports_list, &
                prev_ports_memb);
            struct neighbor *ne = list_new(
                prev_ports_list, &prev_ports_memb);
            ne->addr = msg.port;
            ne->timeouts = 0;
```

```
      } else if (msg.type == ST_WEAK) {
         rimeaddr_copy(&prev, &rimeaddr_node_addr);
         prev_dist = 0;
         list_clear(prev_ports_list, &
            prev_ports_memb);
      }
70  }
   //if (msg = prev) and (mytype = strong)
   if (rimeaddr_cmp(&msg.current_leader, &prev) &&
      msg.type == ST_STRONG && !ist_in_prev(&msg.
      port)  && !rimeaddr_nil(&prev) && !
      rimeaddr_nil(&msg.current_leader))        {
      struct neighbor *ne = list_new(prev_ports_list
         , &prev_ports_memb);
      rimeaddr_copy(&ne->addr, &msg.port);
      ne->timeouts = 0;
   }
   //if (msg >_lexic prev) and (p in prev_ports)
   if (!rimeaddr_lexi_le(&msg.current_leader, &prev,
       msg.cl_dist, prev_dist) && ist_in_prev(&msg.
      port)  && !rimeaddr_nil(&prev) && !
      rimeaddr_nil(&msg.current_leader)) {
      struct neighbor *n3;
80      for(n3 = list_head(prev_ports_list); n3 !=
           NULL; n3 = list_item_next(n3)) {
         if(rimeaddr_cmp(&n3->addr, &msg.port)) {
           break;
         }
       }
      if (n3 != NULL) {
         list_remove(prev_ports_list, n3);
         memb_free(&prev_ports_memb, n3);
      }
   }
90 //Loesche nicht mehr vorhandenen Ports aus
      prev_ports
   struct neighbor *n;
   struct neighbor *nold;
   n = list_head(prev_ports_list);
   while (n != NULL) {
      nold = n;
      n = list_item_next(n);
      if(!ist_alive(&nold->addr)) {
         #if (MYMYDEBUGLEVEL >= 3)
            printf("ST: 22 Loesche node %d.%d aus
               prev_ports\n", nold->addr.u8[0], nold
               ->addr.u8[1]);
100      #endif
         list_remove(prev_ports_list, nold);
         memb_free(&prev_ports_memb, nold);
         }
   }
   //if (prev_ports = 0)
```

```
      if (list_length(prev_ports_list) == 0) {
         prev = rimeaddr_node_addr;
         prev_dist = 0;
         #if (MYMYDEBUGLEVEL >= 2)
            printf("ST: 24\n");
         #endif
      }
      //if (parent is not alive)
      if (!ist_alive(&parent) && !rimeaddr_nil(&parent)
         ) {
         rimeaddr_copy(&current_leader, &
            rimeaddr_node_addr);
         parent.u8[0] = 0;
         parent.u8[1] = 0;
         #if (MYMYDEBUGLEVEL >= 2)
            printf("ST: 25\n");
         #endif
      }

   }
   return 0;
}
#endif
```

Listing C.5: messagepassing.h

```
#ifndef MESSAGEPASSING_H
#define MESSAGEPASSING_H

#define MAX_WRITERS 16

static struct etimer et3;
  static struct etimer et4;
  static struct etimer nutzalgo;
enum {
   NO_PRIV,
   PRIV
};

static uint8_t mode = NO_PRIV;
static uint16_t q_temp;
static uint16_t req_ts;
static uint16_t max_ts;
static uint16_t q = 0;
static uint16_t wert = 0;

/*******        Strukturen und Funktionen
   *********/
struct mp_neighbor {
  struct neighbor *next;    //Wird fuer Contiki-Liste
     benoetigt
  rimeaddr_t addr;
  uint8_t grant;
  uint16_t ts;
```

```
  uint16_t value;
};

MEMB(writers_memb, struct mp_neighbor, MAX_WRITERS);

LIST(writers_list);

MEMB(readers_memb, struct mp_neighbor, MAX_WRITERS);

LIST(readers_list);


struct mp_neighbor* rw_list_new(list_t list, struct
   memb *m5) {
   struct mp_neighbor *n2;
   n2 = memb_alloc(m5);
   list_add(list, n2);
   return n2;
}


struct mp_neighbor* add_writer(uint8_t a, uint8_t b) {
   struct mp_neighbor* w = rw_list_new(writers_list, &
      writers_memb);
   if (w != NULL) {
      w->addr.u8[0] = a;
      w->addr.u8[1] = b;
      w->value = -1;
      return w;
   }
   return NULL;
}

struct mp_neighbor* add_reader(uint8_t a, uint8_t b) {
   struct mp_neighbor* r = rw_list_new(readers_list, &
      readers_memb);
   if (r != NULL) {
      r->addr.u8[0] = a;
      r->addr.u8[1] = b;
      r->value = -1;
      return r;
   }
   return NULL;
}


int writers_granted() {
   struct mp_neighbor *n;
   for(n = list_head(writers_list); n != NULL; n =
      list_item_next(n)) {
         if(n->grant == 0) {
```

```
                return 0;
            }
        }
        return 1;
}

void clear_grants() {
    struct mp_neighbor *n;
    for(n = list_head(writers_list); n != NULL; n =
        list_item_next(n)) {
            n->grant = 0;
    }
}

struct mp_neighbor* get_neighbor(rimeaddr_t *addr) {
    struct mp_neighbor *n;
    for(n = list_head(readers_list); n != NULL; n =
        list_item_next(n)) {
            if(rimeaddr_cmp(&n->addr, addr)) {
                return n;
            }
    }
    return NULL;
}

void update_timestamps() {
    max_ts += 1;
    req_ts = max_ts;
}

int average() {
    uint16_t sum = 0;
    uint8_t anz = 0;
    struct mp_neighbor *n;
    for(n = list_head(readers_list); n != NULL; n =
        list_item_next(n)) {
        if (n->value != -1) {
            sum += n->value;
            anz++;
        }
    }
    sum += q;
    anz++;
    if (anz == 0)
        return 0;
    return (sum / anz);

}

static struct message mp_send;

void messagepassing(struct message msg) {
```

```
if (msg.type == MP_GRANT) {
   if (mode == PRIV && req_ts == msg.ts) {
      struct mp_neighbor *mpn = get_neighbor(&msg.
         from);
      mpn->grant = 1;
      if (writers_granted()) {
         q = q_temp;
         mode = NO_PRIV;
         mp_send.q = q;
         mp_send.ts = req_ts;
         mp_send.type = MP_COMMIT;
         rimeaddr_copy(&mp_send.from, &
            rimeaddr_node_addr);
         send(mp_send); //COMMIT
         etimer_set(&et4, CLOCK_SECOND * 20 +
            random_rand() % (CLOCK_SECOND * 5));
         update_timestamps();
      }
   }

}
if (msg.type == MP_ABORT) {
   if (msg.ts == req_ts)
      mode = NO_PRIV;
}
if (msg.type == MP_COMMIT) {

   struct mp_neighbor *mpn = get_neighbor(&msg.from)
      ;
   mpn->value = msg.q;

   mode = NO_PRIV;
   update_timestamps();
   lcdprint_durchschn(average());
}
if (msg.type == MP_STATE) {

   struct mp_neighbor *mpn = get_neighbor(&msg.from)
      ;
   if (mpn != NULL) {

      if (msg.q != mpn->value) {
         mpn->value = msg.q;

         mode = NO_PRIV;
         update_timestamps();
         lcdprint_durchschn(average());
      }
   }
}
if (msg.type == MP_REQ_COMMIT) {
```

```
      struct mp_neighbor *mpn = get_neighbor(&msg.from)
         ;

      if (mode == NO_PRIV || msg.ts < req_ts) {
         mp_send.ts = msg.ts;
         mp_send.type = MP_GRANT;
         rimeaddr_copy(&mp_send.to , &mpn->addr);
         rimeaddr_copy(&mp_send.from, &
            rimeaddr_node_addr);
         rimeaddr_copy(&mp_send.to, &msg.from);
         send(mp_send);//send GRANT
      } else {
         mp_send.ts = msg.ts;
         mp_send.type = MP_ABORT;
         rimeaddr_copy(&mp_send.to , &mpn->addr);
         rimeaddr_copy(&mp_send.from, &
            rimeaddr_node_addr);
         rimeaddr_copy(&mp_send.to, &msg.from);
         send(mp_send);//send ABORT
      }

   }
   if (mode == NO_PRIV && q != wert && list_length(
      writers_list) > 0 && (random_rand() % 6) == 5) {

      q_temp = wert;
      mode = PRIV;
      clear_grants();
      req_ts = max_ts;

      mp_send.ts = req_ts;
      mp_send.type = MP_REQ_COMMIT;
      rimeaddr_copy(&mp_send.from, &rimeaddr_node_addr)
         ;
      send(mp_send); //REQ_COMMIT
      etimer_set(&et3, CLOCK_SECOND * 8 + random_rand()
          % (CLOCK_SECOND * 5));

   }


}

#endif
```

Listing C.6: routing.h

```
#ifndef ROUTING_H
#define ROUTING_H

rimeaddr_t* in_routing_list(rimeaddr_t* addr) {
   struct neighbor *nn;
```

```
   for(nn = list_head(routing_list); nn != NULL; nn =
      list_item_next(nn)) {
         if(rimeaddr_cmp(&nn->addr, addr)) {
               return nn;
         }
   }
   return NULL;
}


rimeaddr_t* routing(rimeaddr_t* to) {
   struct neighbor *nn;

   nn = in_routing_list(to);
   if (nn == NULL) {
      for(nn = list_head(neighbors_list); nn != NULL;
         nn = list_item_next(nn)) {

            if(rimeaddr_cmp(&nn->addr, to)) {
                  break;
            }
      }
      if (nn == NULL) {
         return &parent;
      } else {
         return &nn->addr;
      }
   } else {
      return &nn->routing_port;
   }
}



#endif
```