Semesterarbeit

Kontrolle und Visualisierung eines Sensornetzwerks

Severin Winkler
email: swinkler@student.ethz.ch

Departement Informatik
ETH Zürich

Prof. Dr. Roger Wattenhofer
Distributed Computing Group
Betreuer: Nicolas Burri, Pascal von Rickenbach

Inhaltsverzeichnis

1 Einführung **3**
1.1 Aufgabenstellung 4
1.2 Aufbau des Netzwerks 5

2 Design und Implementation **7**
2.1 Sensorknoten 7
2.2 PC 11

3 Benutzung **13**
3.1 Übersicht 13
3.2 Starten 13
3.3 Darstellung der Topologie 13
3.4 Topologie 14
3.5 LEDs 15
3.6 Radio 15
3.7 Fehlermeldungen 15

4 Resultate **17**
4.1 Tests 17
4.2 Fazit 17

5 Ausblick **18**

Anhang **20**

A Installation & Konfiguration **20**

B Nachrichtenformate **21**

C Interfaces **21**

1 Einführung

Sensorknoten sind kleine Geräte, die über beschränkte Ressoucen, was Speicherkapazität, Rechenleistung und auch Energieversorgung betrifft, verfügen. Ein Sensornetzwerk wird aufgebaut, indem eine Vielzahl von Knoten über ein Gebiet verteilt wird. Die Knoten können drahtlos miteinander kommunizieren und bilden so selbstorganisierende Netzwerke. Die Sensorknoten erfüllen dann typischerweise Aufgaben wie das Sammeln von Informationen über ihre Umgebung.

Die Entwicklung von Anwendungen für Sensornetzwerke ist eine anspruchsvolle und oftmals aufwändige Aufgabe. Der Umgang mit den beschränkten Ressourcen stellt spezielle Anforderungen bei der Programmierung. Auch die anschliessende Verifikation des korrekten Verhaltens der für Sensornetzwerke implementierten Algorithmen ist schwierig. Die verwendeten Algorithmen sind meist dezentral. Zudem verfügen typische Sensorknoten als einzige Benutzerschnittstelle über drei LEDs. So ist die Sicht auf den Zustand einer Anwendung sehr eingeschränkt. Erschwerend kommen die Unzuverlässigkeit der Verbindungen und nichtdeterministisches Verhalten durch Umwelteinflüsse bei der drahtlosen Kommunikation hinzu. Dadurch ist der Zustand des Netzwerks im allgemeinen nicht stabil. Das führt zu einem komplexen und schwer interpretierbaren Fehlerverhalten.

Um diese Aufgabe zu vereinfachen, können die Sensorknoten relevante Informationen über den Zustand der Anwendung oder des Netzwerks an einen ausgezeichneten Knoten senden. Von dort können sie an einen PC weitergegeben werden und zum Debuggen der Anwendung verwendet werden. Um die Entwicklung zu beschleunigen, sollte die dazu notwendige Funktionalität nicht für jede Anwendung neu programmiert werden müssen. Eine Lösung sind wiederverwendbare (Software-)Module, die diese Funktionalität einer beliebigen Anwendung als Service zur Verfügung stellen. Ein Beispiel für einen solchen Service ist die Ermittlung der Topologie des Netzwerks.

Eine weitere Schwierigkeit stellt das Testen von Anwendungen auf bestimmten und bekannten Netzwerktopologien dar. Zwar können bestimmte Topologien durch Einstellen der Sendestärke und durch entsprechendes Platzieren der Sensorknoten im Raum realisiert werden. Dazu muss aber die aktuelle Topologie des Netzwerks ermittelt werden können. Die Reichweite eines Knoten ist nämlich nicht nur von der (eingestellten) Sendeleistung, sondern auch von andern Faktoren wie der Umgebung oder der aktuell verfügbaren Energie abhängig. Es ist darum schwierig, den Kommunikationsradius eines Knoten exakt vorherzusagen. Durch das Verteilen und Wiedereinsammeln der Knoten ist diese Lösung auch sehr aufwändig. Eine einfache Lösung zu diesem Problem ist die Verwendung eines vollständigen Kommunkationsgraphen und eines in Software implementierten Filters auf den Sensorknoten. Der Filter wird zwischen die Anwendung und die Kommunikationsschicht eingefügt. Er realisiert eine Art logischer Topologie, indem er Nachrichten, die über unerwünschte Verbindungen empfangen werden, nicht an die Anwendung weitergibt. Diese Topologie kann über Nachrichten gesteuert werden. So können beliebige Kommunikationsgraphen simuliert werden, welche stabil sind und sich zur Laufzeit verändern lassen.

1.1 Aufgabenstellung

Das Ziel dieser Arbeit ist der Entwurf und die Implementation eines Frameworks für die Kontrolle und Visualisierung eines Sensornetzwerks. Das Framework sollte in eine beliebige Anwendung für ein Sensornetzwerk eingebunden werden können. Das Framework sollte im Hintergrund der eigentlichen Anwendung laufen und verschiedene Dienste anbieten. Es sollte einem Knoten ermöglichen, periodisch seine Nachbarschaft zu ermitteln und an die Basisstation zu schicken. Hier ist anzumerken, dass der Kommunikationsgraph im Folgenden immer als gerichteter Graph betrachtet wird. Mit der Nachbarschaft eines Knoten v ist die Menge aller Knoten gemeint, die eine gerichtete Kante nach v besitzen. Diese Information sollte auf dem PC visualisiert werden. Weiter sollten vom PC Kommandos an einzelne oder alle Knoten gesendet werden können. Die Kommandos sollten dann von den Knoten interpretiert und ausgeführt werden. Dazu sollten einfache Befehle wie die Steuerung der LEDs oder das Einstellen der Sendestärke erstellt werden. Dann sollte eine Filtermechanismus entworfen werden, der eine beliebige für die Anwendung sichtbare Topologie simuliert. Dazu sollte ein Kommando erstellt werden, das es erlaubt, einzelne Verbindungen aus dieser logischen Topologie zu entfernen und wieder einzufügen. Alle implementierten Kommandos sollten auf dem PC generiert und über die Basisstation an die Knoten versendet werden können. Es sollte darauf geachtet werden, dass die Menge der implementierten Kommandos einfach erweiterbar ist.

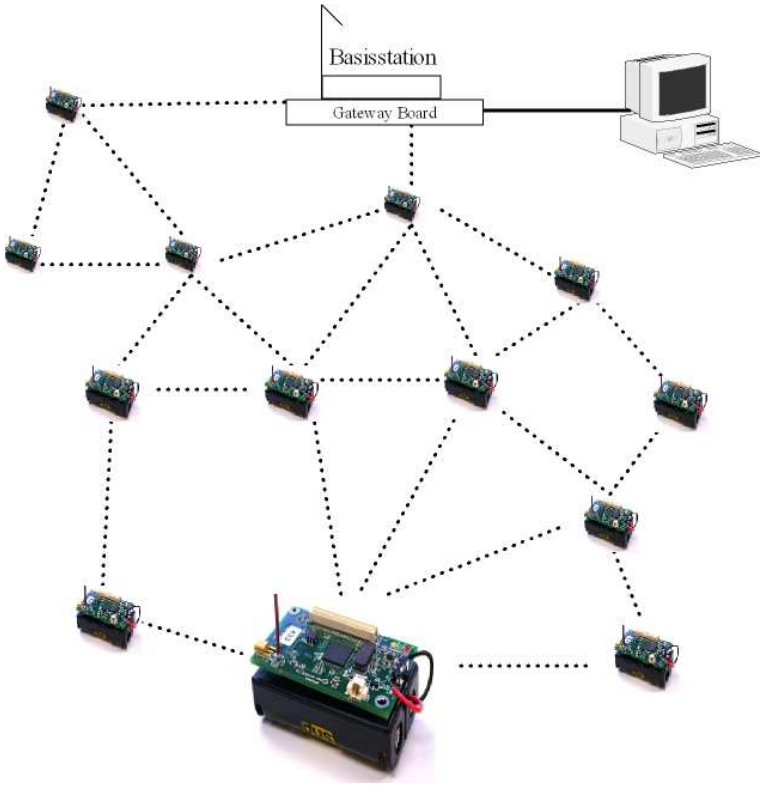

Abbildung 1: Sensornetzwerk

1.2 Aufbau des Netzwerks

Das Framework wurde für ein Netzwerk von Mica2-Knoten, die das Open-Source Betriebssystem TinyOS [2, 3] verwenden, implementiert und getestet. Dabei wurde ein ausgezeichneter Knoten, die Basisstation, über die UART Schnittstelle mit einem Ethernet Gateway (MIB600 [1]), das die Verbindung zum PC herstellt, verbunden.

Auf dem PC wird das in Java geschriebene Interface `MoteIF` verwendet, um über den `SerialForwarder` mit der Basisisstation zu kommunizieren. Der `SerialForwarder` ist eine bestehende Java-Applikation. Sie erlaubt es mehreren Client-Applikationen sich gleichzeitig mit der Basisstation zu verbinden.

Für die Identitäten der Knoten wurden nur 8 Bits verwendet, was für realistische Netzwerke zum Testen von Anwendungen aureichend scheint. Die kürzeren Adressen lassen - wie an den betreffenden Stellen noch genauer erläutert wird - eine effizientere Implementierung zu. Die Adressen '0' und '255' wurden dabei für die Basisstation und die Broadcastaddresse reserviert.

Für die Basisstation wurde die bereits bestehende Anwendung `TOSBase` verwendet, welche über das Radio empfangene Nachrichten über die UART Schnittstelle weiterleitet und umgekehrt. Sie dient also ausschliesslich dazu, die Verbindung zwischen PC und drahtlosem Netzwerk herzustellen. Auf die Möglichkeit, als Basisstation einen Knoten zu verwenden, der selbst Teil des Sensornetzwerks ist und als zusätzliche Funktionalität, die für den PC bestimmten Daten weiterreicht, wurde verzichtet. Dadurch ist eine effizientere Implementierung, der für das Framework verwendeteten Routingalgorithmen möglich. Darauf wird weiter unten noch näher eingegangen.

2 Design und Implementation

2.1 Sensorknoten

Dieser Abschnitt beschäftigt sich mit dem Teil des Frameworks, der auf den Sensorknoten zur Anwendung kommt. Dabei wird vor allem auf die Grundideen eingegangen und versucht, einige Entscheidungen bezüglich des Designs der Anwendung zu motivieren und zu begründen. Details der Implementierung, die für das Verständnis der Anwendung unwichtig sind, werden nicht diskutiert, können aber im Quellcode nachgelesen werden.

Beim Entwurf der grundsätzlichen Struktur des Programms war vor allem die Erweiterbarkeit wichtig. Insbesondere sollte es einfach möglich sein, andere Algorithmen zum Routen der Nachrichten zu verwenden. Die entsprechenden Module sollten problemlos ausgewechselt werden können. Weiter sollte das Hinzufügen zusätzlicher Kommandos unterstützt werden. Eine weitere wichtige Forderung war, dass das Programm in eine beliebige für TinyOS geschriebene Applikation eingebunden werden kann. Damit die Verwendung des Frameworks für die Anwendung möglichst transparent ist, wird eine zusätzliche Schicht zwischen die Applikation und das Kommunkationsmodul `GenericComm` geschoben. Diese Schicht bietet nach oben die gleiche Schnittstelle wie `GenericComm` an. Das heisst, dass sie zum Senden und Empfangen von Nachrichten die Interfaces `SendMsg` und `ReceiveMsg` zur Verfügung stellt. Diese Schicht wird durch die Konfigurationsdatei `LTOP_Comm` repräsentiert, die von der Applikation an Stelle von `GenericComm` als Komponente verwendet wird. Das ist im Normalfall durch die Änderung von wenigen Zeilen in einer Konfigurationsdatei möglich. Die genaue Verwendung von `LTOP_Comm` wird im Anhang A erläutert.

Im Überblick gliedert sich der restliche Aufbau der Anwendung wie folgt: Das Modul `LTOP_CommM`, welches die Funktionalität zu `LTOP_Comm` implementiert, verteilt die ankommenden Nachrichten. Nachrichten, welche für die Applikationsschicht bestimmt sind, werden hier gefiltert. Alle andern Nachrichten werden an ein Routingmodul weitergegeben. Dieses ist dafür zuständig, Kommandos von der Basisstation zu einzelnen oder allen Knoten zu transportieren und Daten von den einzelnen Knoten zur Basisstation zu bringen. Ankommende Kommandos werden vom Dispatcher verarbeitet, indem er sie an die zuständigen Module weitergibt. Das Ermitteln der Nachbarschaft wird vom Modul `Neighbourhood` übernommen. Von dort wird diese Information dann von `LogCommand` ausgelesen, in Nachrichten verpackt und über das Routingmodul an die Basisstation versendet.

Im Folgenden werden die einzelnen Komponenten der Anwendung und ihre Funktionalität kurz erläutert. Einen schematischen Überblick über die wichtigsten Komponenten und ihren Zusammenhang anhand der Schnittstellen wird in Abbildung 1 gegeben.

2.1.1 Dispatcher

In diesem Teil der Anwendung geht es darum, Nachrichten empfangen zu können, deren Inhalt als Befehl zu interpretieren und diesen dann auszuführen. Der hierzu verwendete Mechanisums sollte möglichst einfach erweiterbar sein. Dies lässt sich in TinyOS natürlicherweise mit einem parametrisierten Interface reali-

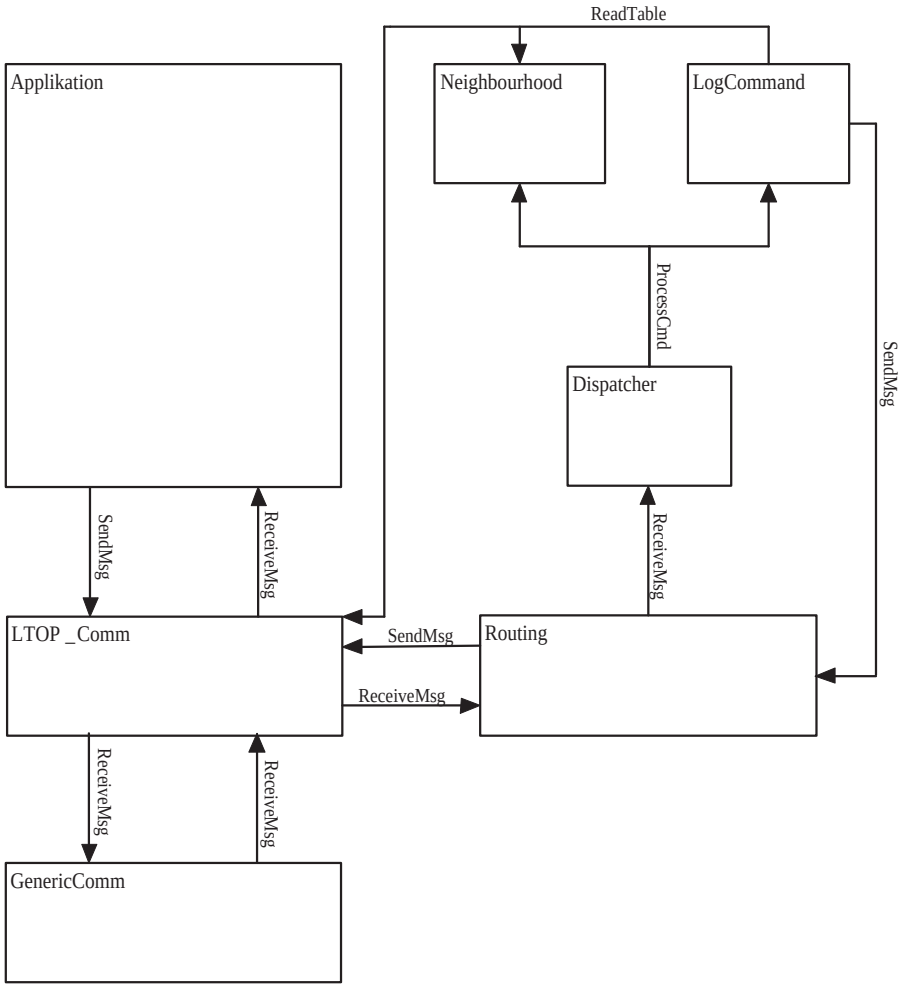

Abbildung 1: Komponenten im Überblick

sieren (vergleiche `Dispatcher Pattern` in [4]). Dazu wird das folgende Interface `ProcessCmd` definiert:

```
interface ProcessCmd{
  command result_t execute(CmdMsgPtr cmd);
  event result_t done(CmdMsgPtr cmd, result_t status);
}
```

Dann wird ein Nachrichtentyp `CmdMsg` definiert (s. Anhang B), der je ein Feld für die Adresse und den Typ des Befehls enthält. Die Adresse der Knoten wird gebraucht, um Unabhängigkeit vom verwendeten Routingalgorithmus zu garantieren. So ist es beispielsweise auch möglich alle Nachrichten durch Fluten zu den Zielknoten zu bringen.

Zusätzliche enthält `CmdMsg` einen Datenteil, der die spezifischen Parameter für die einzelnen Typen von Befehlen enthält und dessen Format für jeden Befehl individuell festgelegt wird. Das Modul Dispatcher empfängt die Nachrichten vom Typ `CmdMsg` und benutzt nun einfach ein Interface `ProcessCmd[uint8_t type]` mit dem Typ des Befehls als Parameter. Um einen neuen Befehl zu implementieren, wird ihm eine eindeutige Identität zugewiesen. Dann wird ein Modul geschrieben, welches das Interface `ProcessCmd` anbietet, um den Befehl auszuführen. Schliesslich wird dieses Interface entsprechend seiner Identität mit dem parametrisierten Interface von `Dispatcher` verbunden. So muss das Modul `Dispatcher` nichts über die implementierten Befehle wissen und ist sehr einfach erweiterbar.

Damit der `Dispatcher` bei der Ausführung von zeitaufwendigeren Kommandos nicht blockiert wird, werden die ankommenden Befehle vom Dispatcher in einem Puffer abgelegt und das Ausführen eines Befehls als Split-Phase-Operation implementiert. Deren Aufruf ist nicht-blockierend und der Dispatcher wird mit dem Event `done` informiert, sobald er den Eintrag im Puffer freigeben kann.

2.1.2 Kontrolle der Topologie

In diesem Teil der Anwendung geht es darum, den weiter oben beschriebenen Filtermechanismus zu realisieren. Dieser soll es ermöglichen, dynamisch Links aus der für die Applikation sichtbaren Topologie zu entfernen und wieder einzufügen. Die Funktionalität hierzu wird in einer Schicht, die zwischen der Applikation und `GenericComm` eingeschoben wird, angesiedelt.

Da es nicht möglich ist, Nachrichten, die an die Broadcastaddresse verschickt werden, bereits beim Sender auszusondern, wird das ganze Filtern erst beim Empfänger vorgenommen. Grundsätzlich wäre es auch möglich, die Aufgabe zwischen Sender und Empfänger zu verteilen. Dazu müssten allerdings auch die Information über die entfernte Verbindung auf beiden Knoten vorhanden sein. Da der Filter über Nachrichten gesteuert wird, könnten Verbindungen dann aber zumindest vorübergehend einen inkonsistenten Zustand annehmen.

Da das TOS_Msg Nachrichtenformat keine Absenderadresse enthält, werden alle von der Applikation versendeteten Nachrichten mit einem zusätzlichen Feld für die Adresse des Senders versehen. Zusätzlich wird auch ein Feld für die ursprüngliche Active-Message-ID der Nachricht eingefügt. Die Active-Messsage-ID einer Nachricht spezifiziert den Handler, der beim Empfang der Nachricht aufgerufen wird. Das hierzu verwendete Format LTOP_TOS_Msg ist in Anhang B definiert. Die Nachrichten werden dann mit der einheitlichen Active-Message-ID

AM_LTOP_TOS_Msg verschickt. Dadurch kann die Applikation den vollen verfügbaren Bereich der Active-Message-IDs verwenden, was der Forderung nach grösstmöglicher Transparenz nachkommt.

Beim Empfänger werden alle ankommenden Nachrichten vom Typ AM_LTOP_TOS_Msg anhand der Absenderaddresse und einer Tabelle mit den entfernten Links gefiltert. Nachrichten, welche den Filter passieren, werden wiederhergestellt, d.h. die zusätzlichen eingefügten Felder werden entfernt und der ursprüngliche Typ wieder eingesetzt.

Durch das Einsetzen von zwei zusätzlichen Feldern gehen allerdings 2 Byte für den Datenteil der Nachricht verloren. Um die standardmässig in TOS_Msg Nachrichten für Daten verfügbaren 29 Byte nicht weiter einschränken zu müssen, wird ein zusätzliche Konstante LTOP_DATA_LENGTH eingeführt. Diese wird von der Applikation an Stelle von TOSH_DATA_LENGTH verwendet und definiert die für die Applikation verfügbare maximale Länge der Daten. Beim Kompilieren der Anwendung kann dann TOSH_DATA_LENGTH - und damit die Länge der verwendeten TOS_Msg Nachrichten - einfach auf (LTOP_DATA_LENGTH + 2) gesetzt werden.

2.1.3 Routing

Zum Routen der Nachrichten werden zwei verschiedene Verfahren verwendet. Ein Multi-Hop Routingverfahren wird gebraucht, um einzelne Knoten adressieren zu können. Damit werden Nachrichten vom PC an die einzelnen Knoten geroutet und Informationen von den Knoten zum PC versendet. Ein Broadcastverfahren erlaubt das Verteilen einer Nachricht an alle Knoten.

Als Multi-Hop Routingverfahren wird hier dynamisches Source Routing verwendet. Die verwendete Implementation beruht weitgehend auf bestehendem Code aus der Diplomarbeit von Yves Weber. Das Verfahren bietet den Vorteil, dass die gesamte Routinginformation beim Ausgangsknoten gespeichert werden kann. Da die Knoten nur Nachrichten an den PC senden, nicht aber an andere Knoten, muss auf einem Knoten nur die Route zur Basisstation gespeichert werden. Da auf der Basisstation `TOSBase` verwendet wird, kann das Routingverfahren auf dem PC implementiert werden. Die Basisstation reicht ankommende Nachrichten an den PC weiter und speist vom PC kommende Nachrichten ins drahtlose Netzwerk ein. Auf dem PC stehen ungleich grössere Ressourcen zur Verfügung. So können dort problemlos Routen zu allen Knoten im Netzwerk gespeichert werden. Der Zustand, der auf den Knoten gespeichert werden muss, bleibt so sehr beschränkt.

Die Implementation verwendet blockierendes Senden mit expliziten Acknowledgements für alle Datenpakete.

Eine Schwierigkeit mit der verwendeten Implementation bestand darin, in den Datenpaketen (`PayloadMsg`, s. Anhang B) genügend Platz für eine sinnvolle Datenmenge zu haben. Da die gesamte Routinginformation in den Datenpacketen mitgesendet wird, mussten dazu die maximale Routenlänge auf 9 Hops und die verwendeten Addressen auf 8 Bits beschränkt werden.

Als Verfahren zum Broadcasten von Nachrichten wird einfaches Fluten verwendet: ein Knoten sendet eine ankommende Nachrichten, die er zum ersten Mal erhält, an alle seine Nachbarn. Um Duplikate zu erkennen, werden pseudozufällig generierte Bitstrings der Länge 16 als Seriennummern verwendet. Die Knoten speichern sich die zuletzt gesehenen Nummern in einer Queue. Durch

das mehrmalige Versenden der Nachrichten ist die Kollisionswahrscheinlichkeit der pseudozufällig erzeugten Strings sicherlich vernachlässigbar gegenüber andern Fehlern bei der Kommunikation. Die Motivation zur Verwendung dieses Verfahrens ist der Umstand, dass es so möglich ist, Nachrichten zu broadcasten, ohne den Zustand der Knoten zu kennen. Nach einem Neustart der Anwendung auf dem PC, kann so der Broadcastkanal verwendet werden, um durch das Versenden eines Reset-Befehls den Zustand des Multi-Hop Routingverfahrens zurückzusetzen.

2.1.4 Detektion der Nachbarschaft

Um den Knoten das periodische Ermitteln ihrer Nachbarschaft zu ermöglichen, versendet jeder Knoten Beacons (`NeighbourMsg`, s. Anhang B) mit seiner Adresse. Jeder Knoten führt dann eine Tabelle mit einem Eintrag für jeden Knoten, von dem er innerhalb eines gewissen vergangenen Zeitfensters ein Beacon erhalten hat. Diese Tabelle im Modul `Neighbourhood` wird periodisch aktualisiert und kann über das Interface `ReadTable` (s. Anhang C) von einem andern Modul ausgelesen werden.

2.1.5 Logging

Das Logging wird im Modul `LogCommand` realisiert. Das Modul lässt sich über Kommandos (`LogCmdMsg`, s. Anhang B) steuern. Es lässt sich mit zwei Modulen über das Interface `ReadTable` verbinden. So kann einerseits die Nachbarschaft aus `Neighbourhood` und andererseits die Tabelle mit allen entfernten Verbindungen aus `LTOP_CommM` gelesen und in Nachrichten verschickt werden.

2.2 PC

Beim Entwurf der pc-seitigen Anwendung stand das Ziel im Vordergrund, eine einfache Benutzeroberfläche zur Verfügung zu stellen. Diese sollte den Zugriff auf die implementierten Funktionen zur Steuerung des Netzwerks zulassen und die von den Knoten gesammelten Informationen verständlich darstellen. So weit wie möglich wurde auch versucht, die Ziele bezüglich Erweiterbarkeit des Frameworks zu unterstützen. Hierzu wird im folgenden auf zwei wichtig erscheinende Punkte bezüglich des Designs eingegangen. Als API für die Entwicklung steht eine mit Javadoc erzeugte Dokumentation zur Verfügung.

2.2.1 Kommunikation

Zur Kommunikation mit der Basisstation wird die bestehende Klasse `MoteIF` verwendet. Diese wird von der Klasse `MHmoteIF` verwendet, welche die benötigten Routingverfahren implementiert. Sie bietet eine Schnittstelle, um Nachrichten über mehrere Hops zu versenden. Um eine Abstraktion von den verwendeten Routingverfahren und den zugehörigen Nachrichtentypen zu erreichen, werden die beiden Java-Interfaces `BasicMessageSender` und `BasicMessageReceiver` verwendet. `BasicMessageSender` ermöglicht das Versenden aller implementierten Kommandos. Über `BasicMessageReceiver` können sich interessierte Listener-Objekte registriern, um Logging-Daten zu erhalten. Um die verwendeten Routingverfahren gegen neue auszutauschen, müssen damit einfach diese beiden Interfaces neu implementiert werden. Um zusätzliche Kommandos zu implemen-

tieren, kann die bestehende Implementation von `BasicMessageSender` erweitert werden.

2.2.2 Darstellung der Topologie

Aus Zeitgründen war es nicht möglich, eine graphische Darstellung der Topologie zu realisieren. Stattdessen wird die Topologie textuell als Adjazenzliste ausgegeben. Das bietet eine, im Gegensatz zu einer Adjazenzmatrix, auch für grössere Graphen relativ übersichtliche Darstellung und ist einfach zu implementieren. Um den Austausch gegen eine alternative Form der Darstellung zu ermöglichen wurde das Observer-Pattern [5] verwendet. Die aktuelle Topologie wird in einem `Topology`-Objekt gespeichert. Dessen Zustand kann von einem beliebigen Objekt, welches das Interface `Topology.Observer` implementiert und sich beim `Topology`-Objekt registriert, ausgelesen werden.

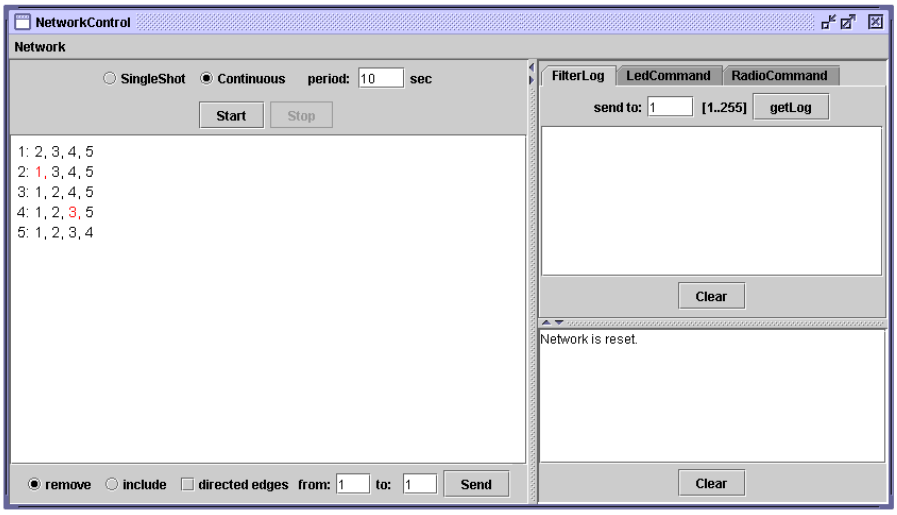

Abbildung 2: Benutzeroberfläche

3 Benutzung

3.1 Übersicht

Die Abbildung 2 gibt eine Übersicht über die Benutzeroberfläche. Der linke Teil dient der Steuerung und Darstellung der Topologie. Rechts oben können verschiedene Felder zum Versenden von Kommandos angewählt werden. Und rechts unten werden Meldungen an den Benutzer ausgegeben.

3.2 Starten

Nach der Konfiguration und Installation gemäss Anhang A, werden die Sensorknoten und die Basisstation gestartet. Dann wird der `SerialForwarder` mit der Basisstation verbunden. Schlisslich wird mit

```
java net.tinyos.tools.control.NetworkControl
```

das GUI gestartet, das sich dann mit dem SerialForwarder verbindet. Beim Starten von `NetworkControl` wird ein Reset-Kommando gebroadcastet, das den Zustand der Routingsschicht auf den Sensorknoten zurücksetzt und gegebenenfalls das Versenden von Logging-Daten stoppt. Dann werden die Filtertabellen der Knoten ausgelesen und die entsprechende logische Topologie daraus konstruiert. So kann die Anwendung jederzeit neu gestartet werden kann, ohne die Sensorknoten zu rebooten.

3.3 Darstellung der Topologie

Die Topologie des Netwerks wird in der Mitte links als Adjazenzliste ausgegeben. Dazu wird für jeden Knoten auf einer eigenen Zeile zuerst die Adresse des Knoten und dann, durch einen Doppelpunkt getrennt, eine Liste seiner Nachbarn in aufsteigender Reihenfolge ausgegegeben. Die den aus der logischen Topologie

entfernten Kanten entsprechenden Knoten werden dabei rot gefärbt. In der Abbildung 2 ist ein vollständiger Graph mit fünf Knoten dargestellt, bei dem die Kanten (1,2) und (3,4) aus der logischen Topologie entfernt wurden.

3.4 Topologie

Zur Überwachung der Topologie stehen zwei Möglichkeiten zur Verfügung. Wenn die Option 'SingleShot' gewählt wird und dann der 'Start'-Knopf gedrückt wird, wird ein einmaliger Schnappschuss der aktuellen Topologie erzeugt. Die eruierte Topologie wird dabei laufend ausgegeben. Der 'Start'-Knopf bleibt deaktiviert, solange die Berechnung im Gange ist. Um die aktuelle Topologie fortlaufend von den Knoten zu erhalten, wird zuerst 'Continuous' gewählt und ein Intervall in Sekunden im Feld 'period' eingegeben. Ein Intervall von x Sekunden bedeutet dabei, dass die Knoten ihre gesamte Nachbarschaftsinformation alle x Sekunden versenden. Dann wird der 'Start'-Knopf gedrückt. Der 'Start'-Knopf bleibt deaktiviert bis die Überwachung durch Drücken von 'Stop' angehalten wird.

Zur Steuerung der logischen Topologie, d.h. zum Entfernen und Einfügen von Verbindungen, wird zuerst der Startknoten u im Feld 'from' und des Endknotens v im Feld 'to' gewählt. Die Adressen können entweder direkt eingegeben werden oder durch Doppelklicken auf die entsprechenden Adressen in der Anzeige der Topologie angewählt werden. Falls die Option 'directed edges' nicht gewählt wird, werden die beiden gerichteten Kanten (u, v) und (v, u) entfernt, resp. eingefügt, sonst nur (u, v). Durch Drücken von 'Send' werden die entsprechenden Kommandos versendet. Entfernte Kanten werden dann in der Anzeige der Topologie rot gefärbt. Die Anzeige wechselt die Farbe einer Kante erst, wenn die Kommunikationsschicht Acknowledgements für die versendeten Nachrichten erhalten hat.

Eine Hilfe zur Kontrolle bietet das Feld 'FilterCommand' oben rechts in der Benutzeroberfläche. Durch Eingabe einer Adresse und das Drücken von 'getLog' kann man die Tabelle mit allen auf dem entsprechenden Knoten entfernten Verbindungen erhalten. Weiter kann die gesamte lokal gespeicherte Information über die logische Topologie gelöscht werden und die Filtertabellen auf den Knoten neu ausgelesen werden. Dazu muss 'Get Logical Topology' im Menü 'Network' gewählt werden.

3.5 LEDs

Abbildung 4 zeigt das Feld zur Steuerung der LEDs der Sensorknoten. Mit der Option 'set' können die auf den Knoten verfügbaren drei LEDs auf einen bestimmten Wert zwischen 0 und 7 gesetzt werden. Die LEDs können aber auch mit den übrigen Optionen einzeln ein- oder ausgeschaltet werden. Diese Kommandos können beim Debuggen von Anwendungen hilfreich sein.

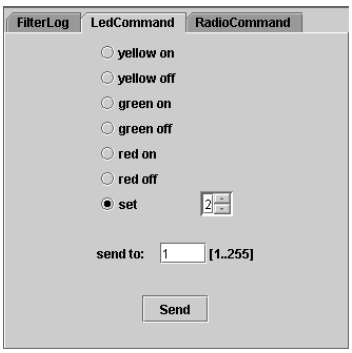

Abbildung 3: Feld zur Steuerung der LEDs der Sensorknoten

3.6 Radio

Abbildung 4 zeigt das Feld zur Steuerung der Sendestärke. Für die Eingabe 'Value', die Sendestärke, sind für Mica-Knoten Werte von 0 (=minimale Stärke) bis 99 (= maximale Stärke) möglich.

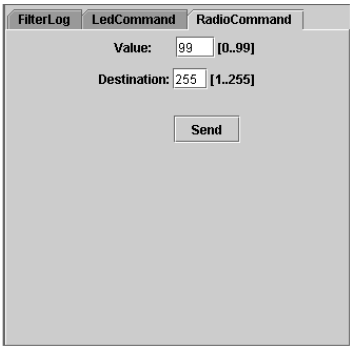

Abbildung 4: Feld zur Steuerung der Sendestärke der Sensorknoten

3.7 Fehlermeldungen

Bei Kommunikationsfehlern beim Versenden von Kommandos an einen einzelnen Knoten gibt es zwei mögliche Fehlermeldungen.

- `Unable to send msg: no route to ADDRESS detected`:
  Diese Meldung bedeutet, dass keine Route zum Zielknoten gefunden wurde. Es ist allerdings möglich, dass die Nachricht den Zielknoten über eine alte Route noch erreicht hat, das zugehörige aber Acknowledgement verloren gegangen ist und erst dann keine (neue) Route mehr gefunden wurde.

- `sending message to node ADDRESS: message not ack'ed`:
  Diese Meldung bedeutet, dass eine Route zum Zielknoten gefunden wurde und das Kommando über diese Route versendet wurde, aber kein Acknowledgement für die Nachricht angekommen ist. Entweder ist die Nachricht selber oder das zugehörige Acknowledgement verloren gegangen.

4 Resultate

4.1 Tests

Während dem Entwicklungsprozess wurden die für TinyOS geschriebenen Module mit TOSSIM [6] getestet. Zum Einspeisen von Packeten in das simulierte Netzwerk wurden für Tython [7] geschriebene Skripte verwendet. Zusätzlich wurden die einzelnen Module auch auf realen Single-Hop Netzwerken getestet. Sinnvolle Tests des gesamten Frameworks wurden erst mit der Fertigstellung der pc-seitigen Anwendung möglich. Auch hier wurde zuerst in einem Single-Hop Netzwerk getestet. Am Ende dieses Testprozesses lief die Anwendung in dieser Umgebung problemlos und auch über mehrere Stunden hinweg stabil. Die gesamten Funktionen verhielten sich für die verwendeten Testfälle korrekt.

Zum Schluss wurde das gesamte Framework in einem Multi-Hop Netzwerk mit 15 Sensorknoten getestet. Dazu wurden die Sensorknoten so im Raum verteilt, dass Verbindungen über mehrere Hops von der Basisstation zu einzelnen Knoten erzwungen wurden. Dabei war es aber nicht möglich, die Sensorknoten so zu platzieren, dass alle Verbindungen im Netzwerk stabil blieben. Mit den höheren Verlustraten bei der Übertragung von Nachrichten erwies sich die verwendete Implementation des dynamischen Source Routings als nicht sehr effizient. Durch eine geeignete Wahl der im Algorithmus verwendeteten Parameter könnte die Effizienz sicherlich noch optimiert werden. Weitere Tests wurden dann aber nicht unternommen - einerseits aus Zeitgründen, aber auch, weil der Fokus der Arbeit nicht auf dem Multi-Hop Routing lag und in einer möglichen Weiterentwicklung das verwendete Routingverfahren ersetzt werden könnte.

4.2 Fazit

Die vorliegende Arbeit war ursprünglich als Teil einer Doppelsemesterarbeit geplant. Der Fokus dieses Teils lag dabei auf der Kontrolle des Netzwerks, insbesondere der Steuerung der logischen Topologie. Der zweite Teil der Arbeit, der sich mit dem Logging von Daten und deren Visualisierung beschäftigt hätte, ist aber nicht zu Stande gekommen. Um trotzdem eine funktionsfähige und sinnvolle Anwendung zu erhalten, wurde die Aufgabenstellung der vorliegenden Arbeit erweitert. Aus Zeitgründen wurde das Logging auf die für die Darstellung der Topologie nötigen Daten eingeschränkt. Das Resultat ist ein Framework, welches in eine beliebige Anwendung für ein Sensornetzwerk eingebunden werden kann. Es bietet eine einfache Benutzeroberfläche, welche die Topologie des Netzwerks darstellen kann und die Steurung einer simulierten Topologie erlaubt. Daneben können auch einfache Kommandos zur Steuerung der LEDs oder der Sendestärke an die Knoten versendet werden.

5 Ausblick

Im Folgenden werden einige Punkte, welche für eine Weiterentwicklung und Verbesserung des implementierten Frameworks wichtig scheinen, kurz erläutert.

Routing Für eine Verwendung des Frameworks in Multi-Hop-Netzwerken wäre ein effizienteres Routingverfahren wünschenswert. Eine Verbesserung würde möglicherweise ein Verfahren bringen, welches sowohl zuverlässige Übermittlung von Nachrichten als auch Übermittlung nach dem Best-Effort-Prinzip erlaubt. Letzteres könnte zum Beispiel für das Versenden der Nachbarschaftslisten an den PC verwendet werden. Für das Versenden von Kommandos könnte dann weiterhin die zuverlässige Übermittlung gebraucht werden.

Logging Die Möglichkeiten zum Logging sind auf Informationen über die Topologie beschränkt und lassen sich nur schwer erweitern. In einer Weiterentwicklung des Frameworks sollte ein Mechanismus gefunden werden, der leichter erweiterbar ist. So könnten auch andere Daten, welche zum Debuggen von Anwendungen hilfreich sein könnten, eruiert werden. Hierzu wäre wiederum ein verbessertes Routingverfahren dienlich.

Benutzerfreundlichkeit Auf Seite des PCs gibt es sicherlich zahlreiche Möglichkeiten zur Verbesserung der Benutzerfreundlichkeit. Die Darstellung der Topologie könnte graphisch erfolgen. Dabei könnten auch zusätzliche Informationen über den Zustand der Knoten, wie die Sendestärke oder die Anzahl gesendeter Packete, in die Darstellung integriert werden.

Das Erzeugen einer logischen Topologie im Netzwerk ist ziemlich aufwändig, weil man ausgehend von einem vollständigen Kommunikationsgraphen alle unerwünschten Verbindungen einzeln entfernen muss. In einer Weiterentwicklung könnte ein Dateiformat zur Beschreibung einer Topologie definiert werden. Das Programm sollte dann eine Datei in diesem Format interpretieren und die entsprechende Topologie automatisch erzeugen können.

Referenzen

[1] Crossbow Technology: http://www.xbow.com/

[2] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, MattWelsh, Eric Brewer, and David Culler. *TinyOS: An operating system for wireless sensor networks.*InW.Weber, J. Rabaey, and E. Aarts, editors, Ambient Intelligence. Springer-Verlag, New York, NY, 2004.

[3] TinyOS Webseite: http://www.tinyos.net/

[4] David Gay, Philip Levis, David Culler, *Software Design Patterns for TinyOS.* In Proceedings of the ACM SIGPLAY/SIGBED 2005 Conference on Languages, Compilers, and Tools for Embedded Systems, ACM Press, 2005.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patters: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] Philip Levis, Nelson Lee, Matt Welsh, and David Culler, *TOSSIM: Accurate and scalable simulation of entire TinyOS applications,* in Proceedings of SenSys, June 2003.

[7] Tython Manual: http://www.cs.berkeley.edu/~pal/research/tython-manual.pdf

Anhang

A Installation & Konfiguration

Entpacke das Archiv `NetworkContorl.zip`.

PC

- Kopiere den Ordner control nach
  `tinyos_directory/tools/java/net/tinyos/tools`

Knoten

1. Der Ordner NetworkControl enthält alle nötigen Dateien für die Einbindung einer Tinyos-Anwendung. Diese Dateien werden in den Ordner kopiert, der die einzubindende Applikation enthält.

2. Die Datei NetworkControl/Application.nc gibt ein Beispiel für die Einbindung der Dummy-Applikation NetworkControl/ApplicationM.nc mit dem zugehörigen Makefile NetworkControl/Makefile.

3. Ersetze in der Anwendung `TOSH_DATA_LENGTH` durch `LTOP_DATA_LENGTH`. Um die normale Datenlänge von 29 zu nutzen, muss `TOSH_DATA_LENGTH` auf 31 gesetzt werden, indem MSG_SIZE=31 ins Makefile eingefügt wird, und `LTOP_DATA_LENGTH` in `NetworkControl/LTOP.h` auf 29 gesetzt werden.

4. Setze die Konstante `MAX_NEIGHBOUR` in `NetworkControl/LTOP.h` auf eine obere Schranke für die maximale Anzahl benachbarter Knoten, die ein Knoten verwalten können muss. Ein tieferer Wert lässt effizienteres Senden und Empfangen von (Applikations-)Nachrichten zu und spart geringfügig Speicher auf den Knoten.

5. Kompiliere die Anwendung

6. Installiere die Anwendung auf allen Sensorknoten

7. Installiere `TOSBase` auf dem Knoten, der als Basisstation dient. Falls die Nachrichtenlänge verändert wurde, muss auch TOSBase mit angepasstem Makefile neu kompiliert werden.

B Nachrichtenformate

```
typedef struct CmdMsg {
    uint8_t addr;
    uint8_t type;
    int8_t data[CMDMsg_DATA_LENGTH];
} CmdMsg;

typedef struct LTOP_TOS_Msg {
    uint8_t sender;
    uint8_t original_type;
    int8_t data[LTOP_DATA_LENGTH];
} LTOP_TOS_Msg;

typedef struct PayloadMsg {
 uint8_t s;                          //for debugging only
 uint8_t source;
 uint8_t dest;
 uint8_t serial;
 uint8_t length;
 uint8_t route[MAX_ROUTE_LENGTH]; //Default: MAX_ROUTE_LENGTH = 8
 uint8_t data[PAYLOAD_LENGTH];    //Default: PAYLOAD_LENGTH = 16
} PayloadMsg;

typedef struct NeighbourMsg{
  uint8_t sender;
} NeighbourMsg;
```

C Interfaces

```
interface ReadTable{
  command uint8_t* getTable();
  command int8_t getSize();
}

interface ProcessCmd{
  command result_t execute(CmdMsgPtr cmd);
  event result_t done(CmdMsgPtr cmd, result_t status);
}
```