Institut für Verteilte und Parallele Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 2236

Entwicklung eines Systems zur dynamischen Installation von Komponenten in Sensornetzwerken

Robert Sauter

Studiengang:	Informatik
Prüfer:	Professor Dr. Kurt Rothermel
Betreuer:	Dr. Pedro José Marrón
begonnen am:	1. Juli 2004
beendet am:	25. Januar 2005
CR-Klassifikation:	C.2.2, C.3

Kurzfassung

Technologische Fortschritte haben in den letzten Jahren die Entwicklung von drahtlosen Sensornetzwerken, bestehend aus mit Sensoren bestückten Miniaturcomputern, die miteinander drahtlos kommunizieren können, möglich gemacht. Von diesen Netzwerken wird erwartet, dass sie u.a. die Beobachtung und Analyse von Umweltprozessen ohne ständige menschliche Überwachung und mit einer minimalen Beeinflussung der beobachteten Prozesse möglich machen. Die Ressourcenknappheit der Knoten macht aber spezielle Algorithmen und Systeme notwendig.

Mit der zunehmenden Verbreitung lässt sich ein Trend zu Heterogenität dieser Netzwerke beobachten. Die Knoten können sich sowohl anhand ihrer Hardware, etwa ihrer Sensoren oder ihre Rechenleistung, als auch durch ihre Rolle innerhalb des Netzwerks unterscheiden.

In dieser Arbeit wird ein generischer Rollezuordnungsalgorithmus zur Selbstkonfiguration des Netzwerks entworfen, mit dem sich die Topologie benutzerdefiniert festlegen lässt. Es wird ein System zur dynamischen Installation von Komponenten entwickelt. Auf dem Algorithmus der die Verteilung von Software im Netzwerk erlaubt, liegt der Fokus dieser Arbeit. Dieser Algorithmus benutzt Rolleninformationen, um den Kommunikationsaufwand zu verringern.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation 1

1.2 Aufgabenstellung 3

1.3 Überblick 3

2 Grundlagen **5**

2.1 Überblick 5

2.2 Berkeley Motes: Die Hardware 5

2.2.1 Überblick 5

2.2.2 Mica-2 Motes 6

2.2.3 Evolution 8

2.3 nesC: Die Programmiersprache 8

2.3.1 Überblick 8

2.3.2 Interfaces 8

2.3.3 Module 9

2.3.4 Konfigurationen 9

2.3.5 Nebenläufigkeit 9

2.3.6 Split-Phase Operationen 9

2.3.7 Statisch gegenüber dynamisch 10

2.4 TinyOS: Das Betriebssystem 10

2.4.1 Überblick 10

2.4.2 Kommunikationsprimitive 10

2.5 Tossim: Der Simulator 11

2.5.1 Überblick 11
2.5.2 Einschränkungen 11
2.5.3 Weiterentwicklung 12
2.5.4 TinyViz 12
2.6 Zusammenfassung 13

3 TinyCubus 15
3.1 Überblick 15
3.2 Motivation 15
3.3 Anforderungen 16
3.4 Architekturübersicht 17
3.5 Tiny Data Management Framework 17
3.5.1 Cubus 17
3.6 Tiny Cross Layer Framework 19
3.6.1 State Repository 19
3.6.2 Callbacks 19
3.7 Tiny Configuration Engine 20
3.7.1 Topologiemanager 20
3.7.2 Rollenzuordnungsalgorithmus 20
3.7.3 Codeverteilungsalgorithmus 20
3.8 Einordnung dieser Arbeit 20
3.9 Zusammenfassung 20

4 Anforderungen und Problemfelder 21
4.1 Überblick 21
4.2 Rollenzuordnung 21
4.2.1 Aufgabe 22
4.2.2 Funktionalität 22
4.2.3 Eigenschaften 22
4.3 Codeverteilung 23
4.3.1 Aufgabe 23

4.3.2 Funktionalität 23
4.3.3 Eigenschaften 24
4.4 Wünschenswerte Eigenschaften 24
4.4.1 Konkurrenz dieser Eigenschaften 25
4.5 Grundsätze des Softwareengineering 25
4.6 Zusammenfassung 25

5 Tiny Configuration Engine 27
5.1 Überblick 27
5.2 Architektur 27
5.3 1-Hop-Übertragung (1-hop-Transmit) 29
5.3.1 Anforderungen an den 1-Hop-Übertragungsalgorithmus . . 29
5.3.2 Hierarchische Aufteilung der Daten 29
5.3.3 1-n-Übertragung 30
5.3.4 Sitzungen 30
5.3.5 Datennachrichten 30
5.3.6 Kontrollnachrichten 30
5.3.7 Timeouts 30
5.3.8 Prüfsummen 31
5.3.9 Kein Pipelining 31
5.4 Dateisystem (RAFS) 31
5.4.1 Überblick 31
5.4.2 Anforderungen 32
5.4.3 Verfügbare Implementierungen 33
5.4.4 Konzepte 34
5.4.5 Architektur 34
5.4.6 Ausblick 35
5.5 Persistenz (DistributionStorage) 36
5.6 Topologiemanager (TopologyManager) 36
5.7 Zusammenfassung 36

6 Generischer Rollenzuordnungsalgorithmus 39
6.1 Überblick . . . 39
6.2 Verwandte Arbeiten . . . 39
6.3 Spezifikationssprache . . . 40
6.3.1 Beispiel . . . 40
6.3.2 Relevanz für diese Arbeit . . . 42
6.4 Annahmen . . . 42
6.5 Ablauf . . . 42
6.5.1 Zustände . . . 42
6.5.2 Arbeitsweise . . . 43
6.5.3 Beispiel . . . 44
6.6 Das Schleifenproblem . . . 45
6.6.1 Beispiel . . . 45
6.6.2 Lösungsansätze . . . 46
6.7 Einschränkungen und Nachteile . . . 47
6.8 Vorteile . . . 48
6.9 Ausblick . . . 48
6.10 Zusammenfassung . . . 49

7 Rollenbasierter Codeverteilungsalgorithmus 51
7.1 Überblick . . . 51
7.2 Verwandte Arbeiten . . . 51
7.3 Netzwerkmodell . . . 52
7.4 Annahmen . . . 53
7.5 Ablauf . . . 53
7.5.1 Prinzip . . . 54
7.5.2 Mehrere Gateways . . . 54
7.5.3 Mehrere Rollen . . . 54
7.5.4 Mobile Knoten . . . 54
7.6 Einschränkungen und Nachteile . . . 55
7.7 Vorteile . . . 55
7.8 Ausblick . . . 56
7.9 Zusammenfassung . . . 57

8 Evaluation **59**

8.1 Überblick 59

8.2 Versuchsaufbau 59

8.2.1 Netzwerkstruktur 59

8.2.2 Topologie 60

8.2.3 Annahmen 60

8.3 Zuverlässige Übermittlung einer Nachricht 61

8.3.1 Algorithmus 61

8.3.2 Vollständigkeit 62

8.3.3 Leistung bei der Brückentopologie 62

8.3.4 Leistung bei zufälliger Topologie 64

8.4 Codeverteilung 64

8.4.1 Nachrichtenanzahl 67

8.4.2 Dauer 68

8.4.3 Quadratische Struktur 68

8.5 Zusammenfassung 69

9 Zusammenfassung und Ausblick **73**

9.1 Ausblick 73

9.2 Zusammenfassung 74

Literatur **75**

1. Einleitung

1.1 Motivation

Sensoren sind längst allgegenwärtig. Nicht nur in der Industrie, wo sie etwa aus der Fertigungstechnik oder der Chemiebranche schon lange nicht mehr wegzudenken sind, sondern auch in der Umgebung jedes Einzelnen sind sie zu finden. So ist natürlich jede automatische Tür, ob in Fahrstühlen, Bussen oder Zügen schon lange mit Sensoren ausgestattet. Ein Bereich, in dem sich die zunehmende Verbreitung besonders gut beobachten lässt, ist die Automobilindustrie. Die fortschreitende Verkleinerung und Verbilligung von Sensoren hat dazu geführt, dass in einem Neuwagen heute bis zu 150 verschiedene Sensorchips eingebaut sind, und viele Sicherheitssysteme erst möglich bzw. erschwinglich geworden sind ([Meie04]).

In den letzten Jahren ist es durch fortschreitende Miniaturisierung und Entwicklungen im Bereich der Computerhardware nun möglich geworden, drahtlose Sensornetzwerke (Engl.: Wireless Sensor Networks) aufzubauen. Diese bestehen aus Miniaturcomputern – im Moment sind Größenordnungen einer Streichholzschachtel weit verbreitet –, die mit unterschiedlichen Sensoren bestückt werden können, um Umwelteinflüsse wie Temperatur oder auch Beschleunigung zu messen. Die herausragende Eigenschaft dieser so genannten Sensor Nodes oder auch Motes[1] ist ihre Fähigkeit, miteinander drahtlos zu kommunizieren und so zusammen ein mobiles ad-hoc Netzwerk zu bilden. Zusammen mit einer autonomen Energiequelle ist es möglich, Umweltbedingungen und -veränderungen über einen längeren Zeitraum hinweg aufzuzeichnen und zu analysieren. Das ist ohne ständige Beobachtung durch den Menschen und mit minimaler Beeinflussung der zu beobachtenden Prozessen möglich, was beispielsweise bei der Beobachtung von Ökosystemen wichtig ist.

[1] mote (engl.) – das Stäubchen (Leo Dictionary www.leo.org)

Eine aktive Forschungsrichtung im Bereich der Sensornetzwerke ist die Reprogrammierung der Motes mit neuer Software über das Netz. Da Sensornetzwerke neue Möglichkeiten der Beobachtung und Analyse eröffnen, einerseits durch die geringe Beeinflussung ihrer Umgebung und andererseits durch einen eventuellen Nutzen in unwirtlichen Einsatzgebieten, kann die nötige Funktionalität nicht immer von vorneherein bekannt und implementiert sein. Um die eben genannten Vorteile, geringe Einflussnahme und Robustheit, nicht zu verlieren, ist es nicht möglich Softwareupdates durchzuführen, indem jedes Gerät eingesammelt und neu programmiert wird. Mit der zunehmenden Verbreitung von Sensornetzwerken sind außerdem immer größere Netzwerke mit tausenden von Knoten zu erwarten bei dem eine manuelle Programmierung schlicht undurchführbar ist. Ein System zur Codeverteilung über das Netz und zu dessen Installation ist also jetzt und auch in Zukunft von großer Bedeutung.

Ein Trend im Bereich der Sensornetzwerke ist der Einsatz von unterschiedlicher Hardware. Die Motes können sich einerseits durch ihre Sensoren unterscheiden. Andererseits ist es auch möglich ganz unterschiedliche Geräte zu verwenden, die beispielsweise in ihrer Prozessorleistung stark differieren. Des Weiteren können auch größere Rechner, ganz normale PCs aber auch PDAs, an einem Sensornetzwerk beteiligt sein. Es sind beispielsweise Gateways einsetzbar, über die Sensornetzwerke mit einem normalen LAN oder gar dem Internet verbunden und darüber gesteuert werden können. Außerdem können sich die Motes auch in ihrer Funktion unterscheiden. Das ist einerseits im Bereich der Kommunikation, so können einige Knoten Router-Aufgaben für andere übernehmen, aber auch im „Rechenbetrieb“ möglich, bei dem einige Knoten nur Daten sammeln und andere auch für deren Analyse zuständig sind. Zusammengefasst geht ein Trend in Richtung Heterogenität von Sensornetzwerken.

Um die beschriebene Heterogenität zu formalisieren soll der Begriff *„Rolle“* benutzt werden. Jeder Knoten kann unterschiedliche, auch mehrere, Rollen einnehmen. Dies kann sowohl durch Hardwaregegebenheiten (z.B. unterschiedliche Sensoren), Softwaregegebenheiten (unterschiedliche installierte Komponenten) oder auch durch Ausführung einer Funktion (z.B. Daten sammeln im Gegensatz zu deren Analyse) bestimmt werden. Für den letzten Fall kann die Rolle auch das Ergebnis einer Verhandlung von Sensorknoten sein. So können beispielsweise in einem dicht von Motes besetzten Gebiet einige Daten sammeln (Rolle: „An“), während andere ihren Energiesparmodus aktivieren (Rolle: „Aus“). Nach einer gewissen Zeit können dann unter Berücksichtigung der Batteriestände die Rollen neu verteilt werden.

Der Begriff „Netzwerkstruktur“ wird in dieser Diplomarbeit für den Aufbau eines Sensornetzwerks, d.h. die Position der beteiligten Knoten und ihrer Kommunikationsverbindungen verwendet. Der Begriff „Topologie“ wird für die logische

Organisation des Netzwerks, d.h. die Rollenverteilung und Nachbarschaftsverhältnisse der Knoten benutzt.

1.2 Aufgabenstellung

Im Rahmen des Projekts TinyCubus (s. Kapitel 3 auf Seite 15) wird ein generisches rekonfigurierbares Framework gleichen Namens basierend auf TinyOS (s. Abschnitt 2.4 auf Seite 10), einem Betriebssystem für Sensorknoten, für Berkeley Motes (s. Abschnitt 2.2 auf Seite 5) entwickelt.

In dieser Arbeit soll ein System zur dynamischen Installation von Komponenten entworfen und umgesetzt werden. Der Fokus liegt dabei auf einem energieeffizienten rollenbasierten Algorithmus zur Codeverteilung innerhalb eines Sensornetzwerkes unter Beachtung der besonderen Gegebenheiten wie ausgeprägter Ressourcenknappheit der Motes und einer potentiell große Anzahl und möglicher Mobilität der Knoten. Des Weiteren soll ein Algorithmus zur benutzerdefinierten Festlegung der Topologie eines Sensornetzwerks implementiert werden.

Die ursprüngliche Aufgabenstellung sieht darüber hinaus auch die Konzeption und Implementierung der eigentlichen Installation von neuem Code und der damit einhergehenden Reprogrammierung der Motes vor. Dies hat sich aber als ein so komplexes und umfangreiches Thema erwiesen, dass es in einer weiteren Diplomarbeit [Gaug05] behandelt wird.

Das Gesamtsystem wird einen Teil des TinyCubus Frameworks bilden.

1.3 Überblick

Der Rest dieser Arbeit ist wie folgt aufgebaut. Das nächste Kapitel beschreibt die die verwendete Hard und Softwareplattform. Es folgt die Erläuterung des Gesamtsystems, in das sich diese Arbeit einfügt: das TinyCubus-Framework. Nach den Anforderungen an die zentralen Algorithmen wird das Teilsystem Tiny Configuration Engine erläutert. An die Beschreibung des Rollenzuordnungsalgorithmus und des Codeverteilungsalgorithmus schließt sich die Evaluation an. Das letzte Kapitel fasst diese Arbeit zusammen und es wird ein Ausblick auf potentielle Erweiterungen gegeben.

2. Grundlagen

2.1 Überblick

Nachdem in der Einleitung die Aufgabenstellung erläutert wurde, handelt dieses Kapitel von der Zielplattform dieser Arbeit. Auf einen Überblick über die Zielhardware folgt eine Beschreibung der Implementierungssprache. Zusammen mit dem daraufhin erläuterten Betriebssystem hat sie einen großen Einfluss sowohl auf die Implementierung als auch auf den Entwurf. Es folgt eine Darstellung über den u.a. für die Evaluation verwendeten Simulators.

2.2 Berkeley Motes: Die Hardware

2.2.1 Überblick

Hill et al. unterscheiden in [HHKK04] vier unterschiedliche Klassen von Sensorknoten:

- **Spezialisierte Sensorknoten:** Diese oftmals als „Smart Dust“ bezeichneten hochintegrierte Schaltungen werden jeweils für einen speziellen Zweck entwickelt und eingesetzt. Sie zeichnen sich durch minimale Größe (im Kubikmillimeterbereich), geringste Energievorräte und extreme Ressourcenknappheit aus. Zu ihren Einsatzmöglichkeiten zählt beispielsweise die Verfolgung von Inventargegenständen.

- **Allgemeine Sensorknoten:** Dies ist die Zielplattform für diese Arbeit. Die Knoten haben im Moment häufig die Größe (ohne Antenne) einer Streichholzschachtel und nutzen AA-Batterien als Energiequelle. Es gibt auch schon Geräte etwa in der Größe einer 2-Euro-Münze (allerdings etwa 1 cm hoch).

Die Knoten sind meistens um einen Standard-Mikrocontroller herum aufgebaut und können oftmals je nach Einsatzgebiet mit unterschiedlichen Sensoren ausgestattet werden. Auch diese Klasse zeichnet sich durch Energiemangel und Ressourcenknappheit aus. Für diese Klasse und für die spezialisierten Sensorknoten wird häufig TinyOS als Betriebssystem eingesetzt.

- **Sensorknoten für hohe Datenraten:** Sollen auch Video oder Audio aufgenommen und verarbeitet werden, sind Knoten dieser Klasse nötig. Sie haben jeweils um eine Größenordung höhere Speicherausstattung und Datenübertragungsrate.

- **Gateway Knoten:** Mitglieder dieser Klasse stellen die Verbindung zu LANs oder dem Internet her. Für diese Geräte werden mächtigere Betriebssysteme wie etwa embedded Linux eingesetzt.

Die 2. Klasse der Sensorknoten ist besonders deshalb interessant, weil sie durch ihre geringe Größe und Kosten neue Möglichkeiten eröffnen und in großer Stückzahl eingesetzt werden können. Sie bieten aber noch genug Flexibilität um unterschiedlichste Aufgaben wahrnehmen zu können. Allerdings erfordert ihre Ressourcenknappheit spezielle Kommunikationssysteme und Algorithmen.

2.2.2 Mica-2 Motes

Die Universität Berkeley gehört zu den Pionieren im Bereich der Sensornetzforschung. In Zusammenarbeit mit Intel wurden einige Mote-Typen (Berkeley-Motes) entwickelt, die von der Firma Crossbow Technology vertrieben und weiterentwickelt werden.

Diese Motes haben einige Gemeinsamkeiten. Die Basis aller Typen bildet ein Mikrocontroller der mit wenigen MHz betrieben wird. Eine Eigenschaft, welche das Design der Algorithmen ganz erheblich beeinflusst, ist die Bereitstellung unterschiedlicher Speicherarten. Die eingesetzten Mikrocontroller basieren auf der Harvard-Architektur und unterscheiden zwischen Programm- und Datenspeicher. Besonders die Größe des Datenspeichers ist dabei extrem klein ausgefallen, während der Programmspeicher auch Platz für umfangreichere Anwendungen bietet.

Die primäre Zielplattform für diese Diplomarbeit sind die Mica-2 Motes (s. Abbildung 2.1 auf der nächsten Seite). Sie basieren auf einem Atmel AtMega 128L Mikrocontroller ([Corp]) und bieten 128 kB Programm- und 4 kB Datenspeicher. Dazu kommen ein ebenfalls 4 kB großes EEPROM, das seine Daten auch im ausgeschalteten Zustand behält, sowie 512 kB zusätzlicher Flashspeicher, der allerdings nur seriell angesprochen werden kann. Dieser eignet sich vor allem für das Speichern von Messwerten und behält ebenfalls seine Daten auch im ausgeschalteten Zustand.

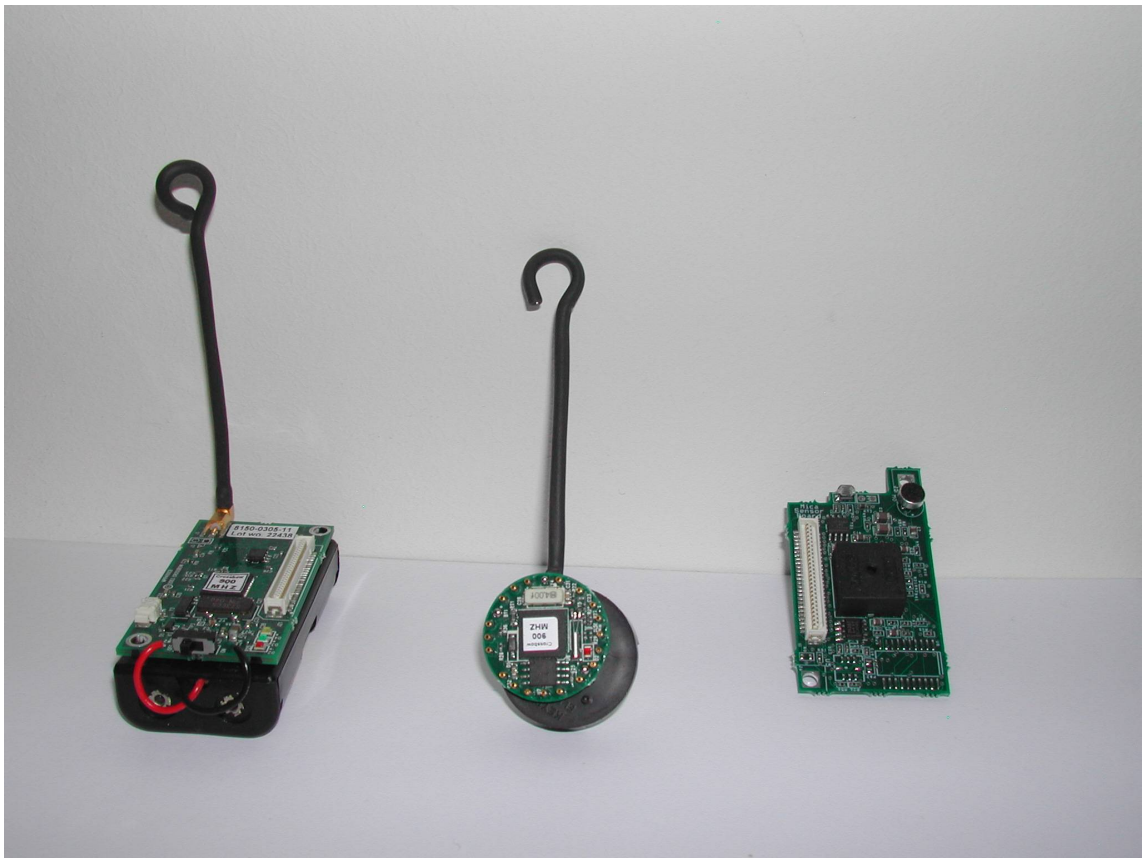

Abbildung 2.1: Photo der verwendeten Motes. Von links nach rechts: mica-2 Mote, mica2dot Mote und ein Sensorboard. (Mit freundlicher Genehmigung von Andreas Lachenmann)

Diese Motes können über Sensorboards mit unterschiedlichen Sensoren, beispielsweise Temperatur, Helligkeit oder Beschleunigung, ausgestattet werden.

Um die Lebenszeit eine Sensornetzwerks zu erhöhen ist besonders auf ein sparsamer Umgang mit Energie zu achten. Aufgrund der Tatsache, dass das Versenden von Nachrichten im Vergleich zu Prozessorberechnungen ein Vielfaches an Energie benötigt ([LeCu02]), muss ein effizienter Kommunikationsalgorithmus vor allem den Nachrichtenaustausch minimieren.

Der verwendete Mikrocontroller bietet eine für das Komponenteninstallationssystem unbedingt erforderliche Möglichkeit: der Programmspeicher ist per Software beschreibbar. Die Tatsache, dass die Reprogrammierung aber mit großen Einschränkungen verbunden ist, so ist der Mikrocontroller etwa in einen speziellen Betriebsmodus zu schalten und Hardwareinterrupts (etwa durch das Funkmodul) sind nur beschränkt verwendbar, ist ein Grund für die Komplexität der Installation von neuen Softwarekomponenten. Diese Komplexität hat wie in der Einleitung schon erwähnt zu einer weiteren Diplomarbeit geführt.

2.2.3 Evolution

In der Folgenden Tabelle (Zusammenfassung aus [HHKK04]) wird ein kurzer Abriss über die Entwicklung der der Motes gegeben.

Mote	**CPU**	**Energieververbrauch Ruhe/Aktiv**	**Speicher RAM/Code**	**E/A und Sensoren**	**Datenübertragungsrate**
Rene 1999	Atmel 8535	0.036 mW 60 mW	512 Byte 8 kB	Sensorboard Konnektor	10 kbps
Mica-2 2001	AtMega 128L	0.036 mW 60 mW	4 kB 128 kB	Sensorboard Konnektor	76 kbps
Telos 2004	Motorola HCS08	0.001 mW 32 mW	4 kB	USB	250 kbps
Mica-Z 2004	AtMega 128L		4 kB 128 kB	Sensorboard Konnektor	250 kbps

Für die Zukunft ist dabei zu erwarten, dass die Entwicklungsfortschritte nach Moore's Law nur teilweise in die Bereitstellung größerer Ressourcen durch die Motes fließen werden. Stattdessen wird wohl ein beträchtlicher Teil in die zunehmende Miniaturisierung und in die Reduzierung des Energieverbrauchs investiert werden, so dass noch auf längere Sicht die Softwareentwicklung auf eine enorme Ressourcenknappheit ausgerichtet werden muss.

2.3 nesC: Die Programmiersprache

2.3.1 Überblick

Die nesC-Programmiersprache ([GLBW+03])ist im Rahmen des TinyOS Projekts entwickelt worden um den Anforderungen, die an die Programmierung der Motes gestellt werden, besser Rechnung zu tragen. Die Erweiterung von C um ein Komponentenkonzept erlaubt eine bessere Strukturierung der Programme. NesC führt neue Konzepte ein, die im Folgenden erläutert werden. Als Ergebnis erstellt der nesC-Compiler einen einfachen C-Quelltext, der dann von einem Standard-C-Compiler in ein ausführbares Format transferiert wird, was eine Vielzahl von Plattformen erreichbar macht, ohne die Komplexität eines Compiler-Backends zu erfordern.

2.3.2 Interfaces

Interfaces sind vergleichbar mit anderen Programmiersprachen wie Java und bieten die Möglichkeit, eine Schnittstelle zu definieren. Das Besondere an den Interfaces von nesC ist dabei ihre Bidirektionalität. Die Fähigkeit nicht nur die angebotenen Funktionen zu deklarieren sondern auch die benutzten Callbacks zu spezifizieren erlaubt eine genauere Definition der Schnittstelle. In nesC werden angebotene Funktionen als „command“ und erforderliche Callbacks als „event“ bezeichnet.

2.3.3 Module

In Modulen wird die eigentliche Funktionalität bereitgestellt. Sie können Schnittstellen sowohl anbieten als auch benutzen. Welche das sind, muss dabei genau spezifiziert werden.

Module sind dabei nicht mit Objekten im Sinne der Objektorientierten Programmierung zu verwechseln, da sie nur genau einmal im Gesamtprogramm vorkommen und nicht instanziiert werden können.

2.3.4 Konfigurationen

Konfigurationen erlauben die Umsetzung eines zentralen Konzepts von nesC: die Trennung von Komponenten und ihrer Zusammensetzung. Deklariert ein Modul, dass es ein bestimmtes Interface benutzt, wird erst in einer Konfiguration diejenige Komponente bestimmt, die über diese Schnittstelle angesprochen werden soll. Die Möglichkeit, auch Konfigurationen Schnittstellen anbieten und benutzen zu lassen, erlaubt die Bildung größerer Blöcke, die aus mehreren Modulen und Konfigurationen bestehen. So besteht etwa das Matchbox Dateisystem ([Gay03b]) von TinyOS aus einer ganzen Reihe von Modulen, die aber nach außen hin durch eine einzige Konfiguration abstrahiert werden.

2.3.5 Nebenläufigkeit

NesC (in Zusammenarbeit mit TinyOS) unterstützt weder Multitasking noch Multithreading. Jede Funktion wird quasi-atomar abgearbeitet, d.h. sie können zwar durch Interrupthandler unterbrochen werden, diese rufen aber keine Anwendungsfunktionen direkt auf. Stattdessen ist es möglich so genannte „Tasks“ in eine Warteschlange zu stellen. Diese werden dann der Reihe nach abgearbeitet. Auch für Anwendungs- und Systemprogramme, die keinen direkten Zugriff auf die Hardware benötigen, gibt es die Möglichkeit Tasks zu benutzen. Das ermöglicht eine Verkürzung der Reaktionszeiten.

2.3.6 Split-Phase Operationen

Da kein Multitasking möglich ist verbieten sich längere blockierende Operationen, etwa das Senden einer Nachricht, um eine angemessene Reaktivität des Gesamtsystems zu gewährleisten. Deshalb sind die meisten TinyOS-Funktionen in zwei Phasen aufgeteilt. Eine Anfrage kehrt sofort zurück, während ihr Ergebnis (beispielsweise eine Sensorauswertung) erst später über ein Ereignis mitgeteilt wird.

2.3.7 Statisch gegenüber dynamisch

NesC ist eine sehr statische Sprache. Es sind beispielsweise keine Funktionszeiger erlaubt (im Moment werden sie nur „missbilligt“), die zur Laufzeit den Programmablauf ändern könnten. Auch ist keine dynamische Speicherverwaltung vorgesehen.

Im vom nesC-Compiler erstellten C-Quelltext werden außerdem alle Funktionen als „static inline“ deklariert. Dies vermindert laut [GLBW+03]nicht nur die Laufzeit sondern auch die Codegröße, allerdings wird dadurch ein dynamischer Austausch von Komponenten unmöglich gemacht, da sie ja nicht mehr voneinander abgegrenzt sind. Für ein System zur dynamischen Installation ist deswegen auch eine Anpassung des nesC-Compilers notwendig.

2.4 TinyOS: Das Betriebssystem

2.4.1 Überblick

TinyOS wurde ursprünglich an der Universität Berkeley entwickelt. Seit der Veröffentlichung als Open Source wird das Projekt aber von einer großen Gemeinschaft und nicht zuletzt von Intel vorangetrieben.

Es ist kein Betriebssystem im herkömmlichen Sinne, dass auf einem System läuft und Anwendungsprogramme starten kann. Stattdessen gibt es eine Vielzahl Komponenten, darunter einige sehr zentrale wie einen Scheduler, von denen eine Auswahl zusammen mit dem Anwendungscode zu einem homogenen Systemimage kompiliert wird.

TinyOS, ursprünglich in C geschrieben, wurde in nesC reimplementiert. Die Laufzeiteigenschaften entsprechen deshalb den oben Erläuterten, wobei die meisten Konzepte zuerst in TinyOS vorhanden waren und dann in der nesC-Programmiersprache formalisiert wurden.

Für diese Arbeit werden nur systemnahe Komponenten wie Timer und die im nächsten Absatz erläuterten Kommunikationsprimitive verwendet.

2.4.2 Kommunikationsprimitive

TinyOS bietet eine einfache Schnittstelle zum Senden und Empfangen von Nachrichten an, wobei der Nachrichtenaustausch nur zwischen benachbarten Knoten funktioniert. Die maximale Größe für Nutzdaten beträgt dabei standardmäßig 29 Bytes. Für manche Mote-Typen gibt es Netzwerkstacks mit Link-Level-Acknowledgements, grundsätzlich wird aber keine zuverlässige Übertragung garantiert.

Damit verschiedene Anwendungen koexistieren können bietet der Netzwerkstack 256 so genannte Active Message IDs. Komponenten registrieren sich (zur Compilezeit) für diese Kennungen, und können dann nur Nachrichten mit einer ihrer

registrierten Kennungen Versenden bzw. werden über den Empfang entsprechender Nachrichten informiert.

Zur Adressierung der Nachrichten an Knoten werden beim Upload der Anwendung auf die Motes manuell IDs vergeben, so dass Nachrichten nur an einen bestimmten Knoten gesendet werden können. Zusätzlich ist natürlich auch ein Broadcast an alle Knoten in Reichweite möglich. Es ist aber kein Multihop-Routing integriert.

2.5 Tossim: Der Simulator

2.5.1 Überblick

Tossim ([LLWC03]) ist ein Simulator für TinyOS. Durch den Austausch systemnaher Komponenten wie Timern und dem Scheduler einerseits, und durch die Unterstützung durch den nesC-Compiler andererseits, ist es möglich nesC-Sourcecode ohne Änderung für PCs zu kompilieren und auszuführen. Die Möglichkeit, Software mit dem Simulator zu testen und dann ohne Codeanpassungen für Motes zu kompilieren, stellt den größten Vorteil dieses Simulators da. Außerdem ermöglicht er u.a. die Evaluation eines Algorithmus mit vielen Knoten und wird deshalb auch für diese Arbeit eingesetzt.

Der Fokus von Tossim liegt dabei auf der bitweisen Simulation des Netzwerkstacks, um auch Low-Level Kommunikationsprotokollen mit ausreichender Genauigkeit simulieren zu können. Zur Konfiguration ist dabei die Bitfehlerrate zwischen je zwei Knoten anzugeben.

2.5.2 Einschränkungen

Leider besitzt Tossim im gegenwärtigen Entwicklungsstand auch einige Einschränkungen, von denen die wichtigsten und für diese Arbeit relevanten im Folgenden aufgeführt sind:

- **Keine Simulation der CPU-Zeit:** Während der Netzwerkstack mit großer Genauigkeit simuliert wird, ist dies für die Ausführung von Berechnungen nicht der Fall. Jede Prozedur wird ausgeführt, ohne auf die Simulationszeit Einfluss zu nehmen. Da allerdings für diese Arbeit keine großen Berechnungen nötig sind, und auch die Zeitmessungen hauptsächlich durch die Dauer des Nachrichtenaustauschs bestimmt werden, ist das keine allzu große Einschränkung.

- **Mangelnde Skalierbarkeit:** Theoretisch sind hunderte oder sogar tausende Motes simulierbar, in der Praxis können aber je nach Komplexität des Algorithmus – entscheidend ist hier vor allem der Umfang des Nachrichtenaustauschs – nur bis zu etwa 100 Knoten in vertretbarer Zeit simuliert werden.

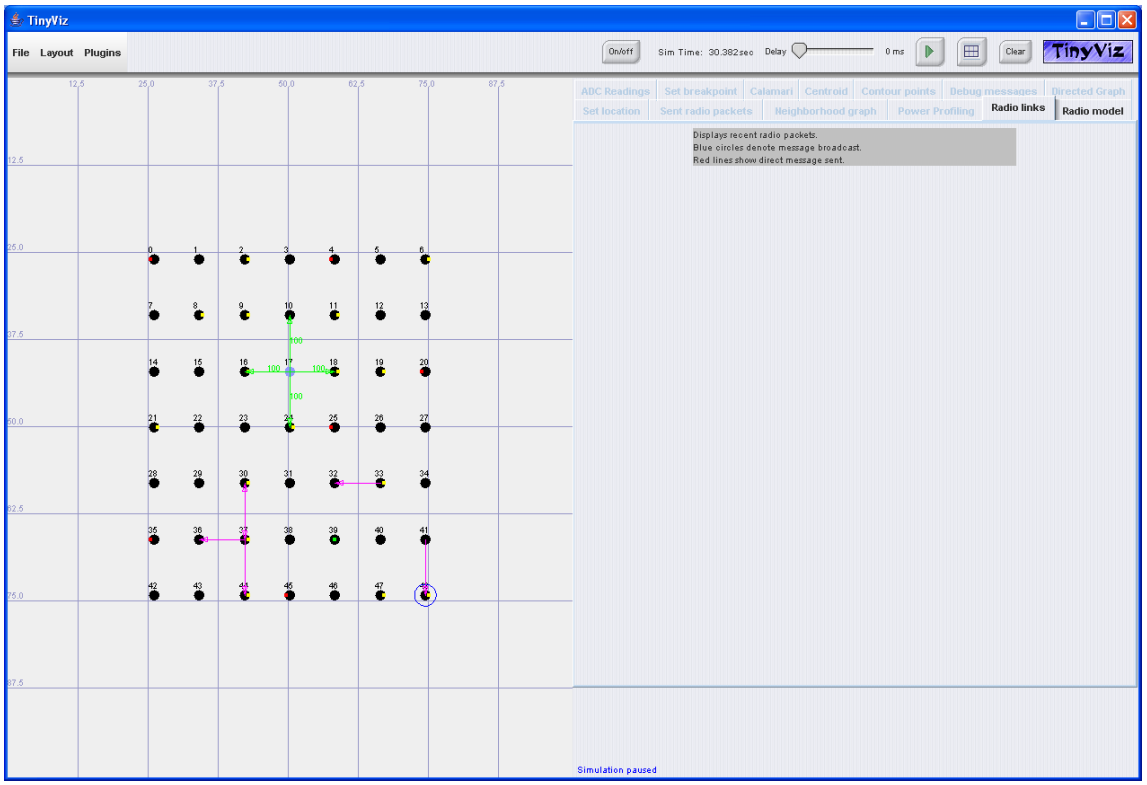

Abbildung 2.2: Screenshot von TinyViz

- **Mangelnde Stabilität:** Auch die Stabilität leidet unter komplexen Algorithmen und großen Knotenzahlen.
- **Nur ein Programm:** Es ist nicht möglich, verschiedene Programme in einer Simulation zu testen, um so heterogene Sensornetzwerke zu simulieren. Um dies trotzdem zu bewerkstelligen müssen manuelle Verzweigungen im Programmcode genutzt werden was den Hauptvorteil von Tossim – ein Quellcode für Simulator und Mote – wieder teilweise zunichte macht.

2.5.3 Weiterentwicklung

Genau wie TinyOS wird Tossim stetig weiterentwickelt. So wird etwa neben einer Bit-Level-Simulation auch an einer Packet-Level-Simulation des Netzwerkstacks gearbeitet. Erste Beta-Versionen lassen auf eine um Größenordnung schnellere Ausführung und damit auch bessere Skalierbarkeit hoffen. Leider ist die Simulation aber noch zu ungenau, um sinnvolle Messwerte zu erhalten.

Eine kurz vor Abschluss dieser Diplomarbeit integrierte Erweiterung ist die Möglichkeit, den Energieverbrauch, besonders des Funkmoduls, zu emulieren. Leider ließ sich diese Möglichkeit nicht mehr in der Evaluation für diese Arbeit nutzen.

2.5.4 TinyViz

TinyViz ([LeLe03]) ist eine Java-Applikation die Tossim-Simulationen zur Laufzeit visualisieren kann. Zur Darstellung greift sie auf eine erweiterbare Menge

von Plugins zurück, die etwa die Fehlerraten zwischen Netzwerkknoten anzeigen können.

TinyViz besitzt inzwischen auch eine Scriptsprache, mit der es möglich ist, beispielsweise im laufenden Betrieb die Positionen der Motes zu verändern und dies auch in geänderten Kommunikationsbedingungen zu reflektieren.

Der Hauptnachteil von TinyViz liegt aber in der starken Herabsetzung der Geschwindigkeit und einer erhöhter Instabilität der Simulation im Vergleich zur alleinigen Nutzung von Tossim. Der Einsatz von TinyViz bietet sich deshalb vor allem am Anfang der Entwicklung an um sich einen Eindruck vom Ablauf eines Algorithmus zu verschaffen.

2.6 Zusammenfassung

In diesem Kapitel wurde die Hard- und Softwareplattform für diese Arbeit vorgestellt. Besonders die Ressourcenknappheit der Motes und die Laufzeiteigenschaften von TinyOS haben Einfluss auf den Entwurf und die Implementierung dieser Arbeit.

Im nächsten Kapitel wird nun das diese Diplomarbeit umgebende Framework vorgestellt, bevor anschließend die Anforderungen an den Rollenzuordnungsalgorithmus und den Codeverteilungsalgorithmus definiert werden.

3. TinyCubus

3.1 Überblick

Diese Arbeit ist Teil des Projekts TinyCubus([MLMH+05] und [Tinya]), welches die Entwicklung eines Frameworks für Sensornetzanwendungen zum Ziel hat. Nach der Darstellung der Plattform sowohl für diese Arbeit als auch für dieses Framework im vorigen Kapitel, soll nun das Framework selbst und die Einordnung dieser Arbeit darin erläutert werden.

Zuerst werden die Motivation und die Anforderungen an das Framework erläutert. Neben der Gesamtarchitektur ist dann vor allem die Beschreibung der drei Hauptkomponenten wichtig. Es folgt die Einordnung dieser Diplomarbeit in das Gesamtprojekt.

3.2 Motivation

Es gibt eine ganze Reihe von Gründen, die für flexibles und adaptives Framework für Sensornetzwerkanwendungen sprechen und teilweise schon in der Einleitung wurden:

- **Steigende Komplexität der Anwendungen:** Mit größeren Sensornetzwerken und Weiterentwicklung der Anwendungen steigt deren Komplexität.
- **Bandbreite der Einsatzzwecke:** Sensornetzwerke lassen sich für eine Vielzahl von Problemen mit teilweise gemeinsamen aber auch unterschiedlichen Anforderungen einsetzen.
- **Evolution der Motes:** Die Hardware entwickelt sich ständig weiter, schon TinyOS ist für mehrere Plattformen verfügbar.

- **Sich ändernde Anforderungen:** Auch während der Lebenszeit eines Netzwerks können sich die Anforderungen und eingesetzten Anwendungen ändern.
- **Heterogene Hardware:** Innerhalb von Sensornetzwerken können schon heute verschiedene Motes oder Konfigurationen (angeschlossene Sensoren) zum Einsatz kommen.
- **Heterogene Funktionalität:** Knoten können auch innerhalb eines Netzwerks unterschiedliche Funktionen wahrnehmen.
- **Ressourcenknappheit:** Die Ressourcenknappheit des Sensorknoten machen Optimierungen der Anwendungen nötig.

Diese Punkte machen die Entwicklung, Optimierung und den Einsatz von Sensornetzwerkanwendungen zu einer schwierigen und zeitaufwendigen Angelegenheit. Für ein Framework, welches diese Komplexität verringert, sprechen also eine ganze Anzahl von Gründen.

3.3 Anforderungen

Ausgehend von zwei sehr unterschiedlichen Projekten der Universität Stuttgart – Sustainable Bridges ([Sust]) und Cartalk 2000 ([RMMM+02]) wurde eine Reihe von Anforderungen an ein generisches Framework eruiert.

- **Sustainable Bridges:** Dieses Projekt hat die kosteneffektive Überwachung von Brücken zum Ziel. Mit einer Reihe fest installierten Knoten sollen strukturelle Schäden festgestellt und gemeldet werden.
- **Cartalk 2000:** Das Ziel dieses Projekts ist die Entwicklung eines Systems zur Unterstützung von Autofahrern. So sollen etwa Warnungen vor Staus, Unfällen oder Fahrbahnverengungen, etwa in Folge von Baustellen, unter den Fahrzeugen ausgetauscht und dem Fahrer zur Verfügung gestellt werden.

Beiden Systemen ist vor allem die Datenzentriertheit und Zustandorientierung gemeinsam. So wechselt beispielsweise die Sustainable-Bridges-Applikation von einem Messmodus, bei dem es darum geht ein aktuelles Ereignis zu beobachten, in einen Analysemodus um die gewonnen Daten auszuwerten.

Ein großer Unterschied liegt in der Mobilität der Knoten im Cartalk Projekt, während die Sustainable-Bridges-Anwendung nur statische installierte Geräte benötigt. Auf der anderen Seite ist ein effektives Energiemanagement bei der Sustainable-Bridges-Applikation nötig, um die Lebensdauer des Netzwerkes zu erhöhen, während die Sensoren im Cartalk 2000 Projekt am elektrischen System des Fahrzeugs angeschlossen sind.

Es lassen sich folgende Anforderungen an ein Framework ermitteln:

- Die Unterstützung der Datenzentriertheit von Sensornetzwerkapplikationen
- Die Unterstützung der Rekonfigurierung der Applikationen
- Die Bereitstellung generischer Komponenten und die Möglichkeit sie für spezielle Zwecke zu parametrisieren
- Die Möglichkeit, anwendungsspezifische Komponenten zu integrieren
- Die Möglichkeit, Komponenten – neue und Updates – energieeffizient über das Netzwerk zu verteilen
- Die Fähigkeit, das System an wechselnde Anforderungen anzupassen und zu optimieren

3.4 Architekturübersicht

Die Abbildung 3.1 auf der nächsten Seite zeigt einen Überblick über die Architektur von TinyCubus. Es wird in voraussichtlich zum größten Teil nesC implementiert und baut auf TinyOS auf. Die Anwendungen interagieren dabei nicht mit TinyOS, sondern registrieren ihre Anforderungen und Dienste mit TinyCubus.

TinyCubus besteht dabei aus drei Teilen, die im Folgenden erläutert werden.

3.5 Tiny Data Management Framework

Das Tiny Data Management Framework bietet eine Reihe von Diensten zur Unterstützung der datenzentrierten Natur von Sensornetzwerkanwendungen an. Dazu gehören beispielsweise Replikation, Caching und Aggregation. Daneben gibt es noch Systemdienste wie etwa zur Zeitsynchronisation. Für jeden Dienst können dabei verschiedene Algorithmen implementiert sein. Das Tiny Data Management Framework ist dann für die Auswahl der geeigneten Implementierung zur Erfüllung der aktuellen Ansprüche zuständig. Diese Komponenten müssen dabei nicht unbedingt bereits vorhanden sein, sondern können auch von anderen Knoten oder Gateways heruntergeladen werden.

Die Klassifikation der Algorithmen sowie der aktuellen Gegebenheiten erfolgt dabei mit Hilfe des „Cubus“.

3.5.1 Cubus

Der Würfel in der Mitte der Abbildung 3.1 auf der nächsten Seite, genannt Cubus, vereinigt drei Klassen von Parametern:

- **Optimierungsparameter,** wie etwa Latenzzeit und Energieverbrauch

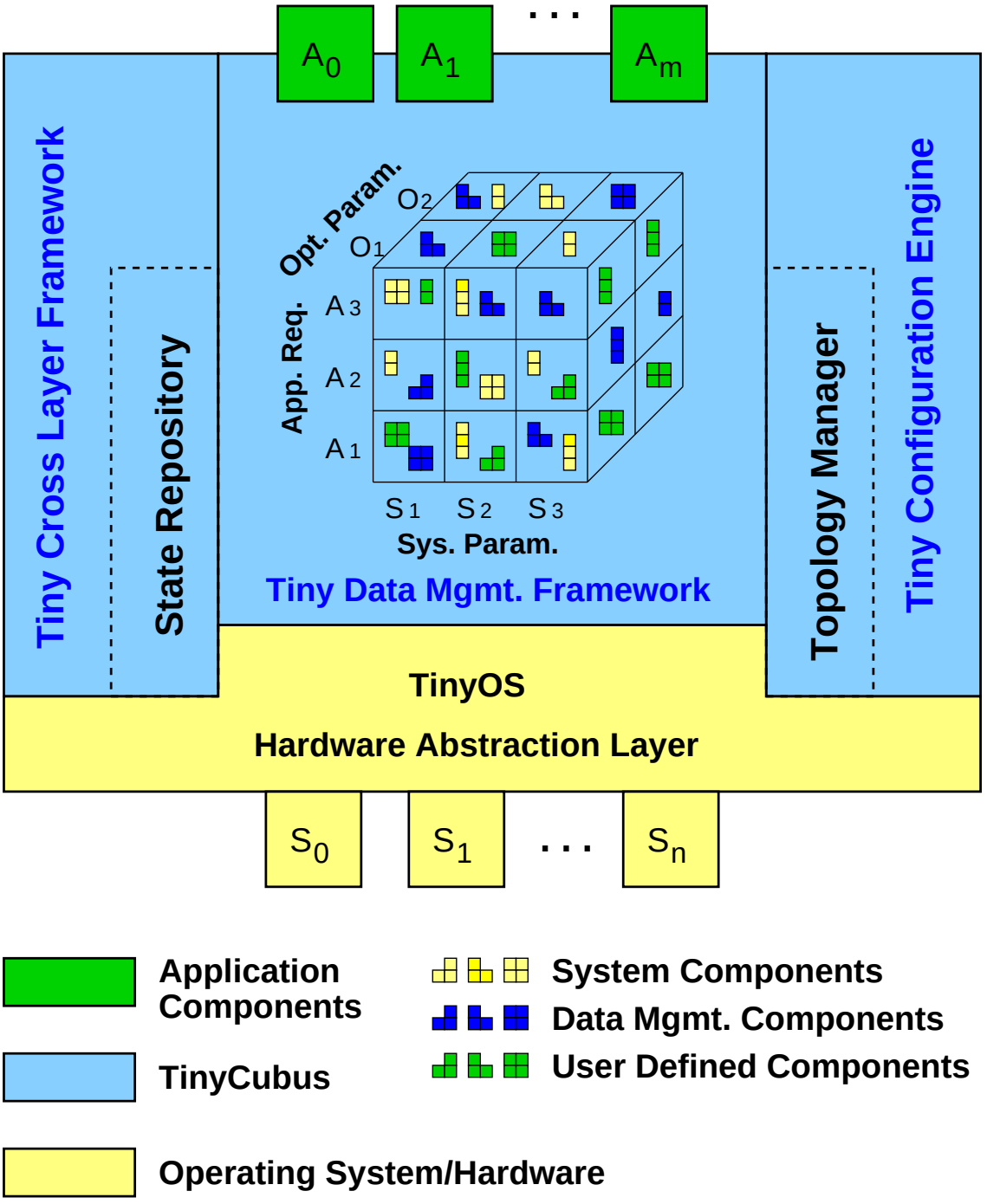

Abbildung 3.1: Die Architektur von TinyCubus (aus [MLMH+05])

- **Systemparameter,** beispielsweise die Mobilität der Sensorknoten
- **Anwendungsanforderungen,** z.B. Genauigkeit und Zuverlässigkeit

Die Anpassung basierend auf den Anforderungen und den verfügbaren Algorithmen ist dabei keine einmalige Angelegenheit, sondern ein Prozess während der gesamten Lebenszeit des Systems.

3.6 Tiny Cross Layer Framework

Die Ressourcenknappheit von Sensorknoten und der dadurch notwendige Zwang zur Optimierung sprechen gegen eine strenge Aufteilung in Schichten. So können beispielsweise sowohl ein Routingalgorithmus als auch eine Anwendungskomponente auf Nachbarschaftsinformationen angewiesen sein, die dann von einer Komponente gesammelt und allen anderen zur Verfügung gestellt werden sollten.

Eine weitere Form der Cross-Layer-Interaktion stellen Callbacks von unteren an höhere Schichten etwa einer Anwendungskomponente dar.

Das Tiny Cross Layer Framework stellt die folgenden Konzepte zur Verfügung.

3.6.1 State Repository

Das State Repository stellt eine zentrale Stelle zur Speicherung aller Cross-Layer-Informationen zur Verfügung. Durch die Nutzung des Tiny Cross Layer Framework als Mediator wird trotz Interaktion zwischen den Schichten die Modularität aufrechterhalten. Außerdem werden so beim Austausch von Komponenten, etwa im Zuge von Optimierungen, bestehende Informationen vorgehalten, die von anderen Komponenten noch benötigt werden und zusätzlich von der neuen Komponente zur Initialisierung benutzt werden können.

Zur Definition welche Daten von Komponenten benötigt und angeboten werden wird die Nutzung einer Spezifikationssprache untersucht. Damit können dann auch Komponenten Informationen nur bei Bedarf anbieten, auch wenn sie sie selber nicht benötigen, was wiederum den Austausch von Algorithmen erleichtert.

3.6.2 Callbacks

Es gibt zwar wie im Abschnitt 2.3.2 auf Seite 8 dargelegt in nesC ein Konzept für Callbacks. Da diese aber nur statisch zur Compilezeit festgelegt werden können, muss zusätzliche Funktionalität für die dynamische Zuordnung durch das Tiny Cross Layer Framework bereitgestellt werden.

3.7 Tiny Configuration Engine

Die zentrale Aufgabe der Tiny Configuration Engine besteht in der Durchführung der eigentlichen Installation. Außerdem obliegt ihr die Verwaltung von Rolleninformationen. Zu diesem Zweck sind drei für die Tiny Configuration Engine zentrale Komponenten vorhanden.

3.7.1 Topologiemanager

Der Topologiemanager ist die zentrale Instanz für die Selbstkonfiguration des Netzwerks. Er bestimmt die Rollen jedes Knoten abhängig von seiner Hardware, Software und Funktion und stellt diese Informationen auch anderen Komponenten zur Verfügung. In einer Weiterentwicklung können auch konkrete Ortsangaben (ermittelt etwa per GPS) gespeichert und verwendet werden.

3.7.2 Rollenzuordnungsalgorithmus

Um die Selbstkonfiguration zu ermöglichen, ist ein Rollenzuordnungsalgorithmus nötig. Neben der Möglichkeit einen Algorithmus komplett manuell zu entwickeln wird auch die Nutzung einer Spezifikationssprache und der Einsatz eines generischen Algorithmus, wie in [RFMB04] beschrieben, untersucht.

3.7.3 Codeverteilungsalgorithmus

Der Codeverteilungsalgorithmus ist für die effiziente Verteilung von Komponenten im Netzwerk zuständig. Dazu ist es beispielsweise möglich, Topologieinformationen auszunutzen. Je nach Cubusparameter sind hier unterschiedliche Algorithmen möglich, die beispielsweise entweder auf Energieverbrauch oder Geschwindigkeit optimiert sind.

3.8 Einordnung dieser Arbeit

Diese Arbeit bildet einen Teil der Tiny Configuration Engine. Neben einem einfachen Topologiemanager und einer Implementierung des generischen Rollenzuordnungsalgorithmus liegt der Fokus vor allem auf einem energieeffizienten Algorithmus zur Codeverteilung. Dazu wird vor allem die Möglichkeit der Nutzung von Rolleninformationen untersucht.

3.9 Zusammenfassung

Dieses Kapitel bietet einen Überblick über das gesamte Framework TinyCubus, und die Stellung dieser Diplomarbeit darin. Während im nächsten Kapitel die Anforderungen an die zentralen Algorithmen der Rollenzuordnung und Codeverteilung erläutert werden, folgt danach eine detaillierte Beschreibung der Tiny Configuration Engine. In den darauf folgenden Kapiteln werden dann die beiden Algorithmen behandelt.

4. Anforderungen und Problemfelder

4.1 Überblick

Nach der Beschreibung des Umfelds dieser Arbeit werden in diesem Kapitel nun die Anforderungen an den Codeverteilungsalgorithmus und an den Rollenzuordnungsalgorithmus spezifiziert. Danach werden noch einige wünschenswerte Eigenschaften erläutert mit denen sich bei gegebener Funktionalität Algorithmen unterscheiden und etwa im Cubus (Abschnitt s. 3.5.1 auf Seite 17) einordnen lassen.

Die Anforderungen beider Algorithmen lassen sich dabei in die folgenden Gruppen einteilen:

- **Aufgabe:** Die Aufgabe beantwortet die Frage, was der Algorithmus eigentlich bewerkstelligt.
- **Funktionalität:** Welche Funktionalität muss er bereitstellen, um seine Aufgabe erfüllen zu können.
- **Eigenschaften:** Die Eigenschaften beschreiben das Verhalten unter externen Bedingungen und deren Änderung.

4.2 Rollenzuordnung

Neben seiner eigentlichen Funktion, bildet der Rollenzuordnungsalgorithmus auch die Grundlage für den nachfolgend beschriebenen rollenbasierten Codeverteilungsalgorithmus.

4.2.1 Aufgabe

Die Aufgabe des Rollenzuordnungsalgorithmus besteht in der Selbstkonfiguration des Netzwerks. Die Knoten sollen selbstständig ihre Rolle einnehmen, um sich einer vorgegebenen Topologie einzupassen.

4.2.2 Funktionalität

- **Generizität:** Es soll ein generischer Algorithmus zur Rollenverteilung entworfen werden, der es erlaubt, die Bestimmung der Rolle eine Knoten und damit die Topologie benutzerdefiniert festzulegen.

- **Einbeziehung von Hardware und Softwaremerkmalen:** Für die Rollenzuordnung ist es nötig, auf Hardwareinformationen, etwa das Vorhandensein eines Sensortyps, aber auch auf die vorhandenen Softwarekomponenten zuzugreifen und diese bei der Rollenbestimmung zu berücksichtigen.

- **Korrektheit:** Die Hauptaufgabe des Algorithmus liegt in der Sicherung des korrekten Ablaufs. Es dürfen keine veralteten Informationen von anderen Knoten verwendet werden. Auch muss die gleichzeitige Rollenbestimmung auf mehreren Knoten, deren Ergebnisse sich beeinflussen können, verhindert werden. Alternativ dazu müssen entstandene Konflikte nachträglich aufgelöst werden.

- **Terminierung:** Für die Terminierung des Algorithmus sind Schleifen – beispielsweise das ständige Springen zweier Knoten zwischen zwei Rollen, der Rollenwechsel des einen bedingt den Rollenwechsel des anderen – ein großes Problem. Schleifen müssen dabei nicht auf zwei Knoten begrenzt sein, sondern können über eine ganze Reihe von Knoten führen. Auch ist ihre Existenz in der Regel nicht nur von der konkreten Rollenspezifikation, sondern häufig auch von der aktuellen Netzwerkstruktur abhängig. Ein konkretes Beispiel wird im Abschnitt 6.6 auf Seite 45 erläutert.

- **Verhalten bei Änderungen des Netzwerks:** Da das „richtige“ Verhalten bei Änderungen, wie dem Ausfall eines Knoten aber auch der Integration zusätzlicher Knoten in das Netzwerk, stark von der gewünschten Topologie und der Anwendung abhängt, liegt auch hier des Fokus vor allem darauf, den Algorithmus an die Erfordernisse anpassen zu können.

4.2.3 Eigenschaften

- **Skalierbarkeit:** Um Skalierbarkeit zu erreichen muss vor allem Lokalität ausgenutzt werden. Falls beispielsweise nur die Rollen der direkten Nachbarn entscheidend sind, dürfen selbstverständlich nicht die Rolleninformationen des gesamten Netzwerks zu einer Entscheidung eingeholt werden.

- **Robustheit:** Unter Robustheit fällt hierbei nur das Verhalten des Algorithmus bei Knotenausfällen oder Kommunikationsstörungen *während* einer laufenden Rollenverhandlung. Das Problem von Änderungen des Netzwerks außerhalb dieses Zeitraums wird oben beschrieben.

- **Unterstützung mobiler Knoten:** Da mobile Knoten aus Sicht der Motes mit einer hohen Fluktuation vergleichbar sind, gilt hier das in den Absätzen „Verhalten bei Änderungen des Netzwerks„ und „Robustheit“ beschriebene Problem. Es ist vor allem die Möglichkeit der benutzerdefinierten Anpassung des Algorithmus notwendig. Eine Definition der Rolle „mobil“ ist aber unbestritten nützlich und kann so auch von anderen Algorithmen, wie etwa der Codeverteilung, zur Anpassung verwendet werden.

4.3 Codeverteilung

Neben der Aufgabe des Codeverteilungsalgorithmus lassen sich fünf Funktionen und drei Eigenschaften des Algorithmus definieren, die im Folgenden erläutert werden.

4.3.1 Aufgabe

Die Aufgabe des Codeverteilungsalgorithmus besteht in der zuverlässigen Verteilung neuer oder geänderter Komponenten - eines Codeabbilds – von einer oder mehreren Quellen an die dafür vorgesehenen Knoten des Netzwerks.

Allgemeiner betrachtet besteht die Problemstellung in der zuverlässigen Übertragung einer großen Datenmenge von wenigen (oft nur einem) an viele Knoten.

4.3.2 Funktionalität

Die folgende Funktionalität muss der Codeverteilungsalgorithmus bereitstellen:

- **Sicherstellen der Integrität der Daten:** Für die Verteilung von ausführbarem Code ist Sicherung der Integrität der übertragenden Daten unverzichtbar, da sonst Abstürze oder fehlerhaftes Verhalten fast unausweichlich sind.

- **Multihop-Routing:** Um ein ganzes Netwerk mit neuem Code zu versorgen, muss der Algorithmus natürlich auch mehrere Knoten überbrücken können.

- **Fortsetzung von Teilübertragungen:** Da mit Störungen in Sensornetzwerken immer gerechnet werden muss und Codeupdates eine für die Ressourcen von Motes erhebliche Größe (mehrere kB) besitzen, ist eine Möglichkeit zur Fortsetzung einer bereits begonnen Übertragung auch unter Nutzung von alternativen Quellen unbedingt notwendig.

- **Unterstützung mehrerer Komponenten:** Da das TinyCubus-Framework eine Vielzahl von Komponenten und deren Austausch unterstützen soll, müssen auch einzelne Komponenten – im Gegensatz zum gesamten Systemimage – übertragen und auf den Sensorknoten gespeichert werden können.
- **Rollenspezifische Verteilung:** Während die obigen Anforderungen eigentlich an jeden Codeverteilungsalgorithmus gestellt werden können, liegt der Fokus dieser Arbeit auf der Nutzung von Rolleninformationen, um die Verteilung zu steuern und effizient zu gestalten.

4.3.3 Eigenschaften

Die folgenden Eigenschaften soll der Codeverteilungsalgorithmus besitzen:

- **Robustheit:** Die Knoten von Sensornetzen unterliegen einer hohen Fluktuationsrate. Neben dem kompletten Ausfall von Geräten, muss auch mit Störungen des Kommunikationskanals gerechnet und der Algorithmus im Hinblick darauf entwickelt werden.
- **Skalierbarkeit:** Mit der weiteren Verbreitung von Sensornetzwerken sind auch Installationen mit mehreren Hundert, Tausend oder gar Zehntausend Knoten zu erwarten. Deshalb sollte der Codeverteilungsalgorithmus auch gute Skalierungseigenschaften besitzen.
- **Unterstützung mobiler Knoten:** Der Algorithmus sollte auch bei mobilen Knoten funktionieren. Dies kann zum Teil schon durch die Erfüllung der Forderungen nach Robustheit und der Unterstützung von Teilübertragungen erreicht werden.

4.4 Wünschenswerte Eigenschaften

Neben den Anforderungen an die Algorithmen, gibt es auch noch einige wünschenswerte Eigenschaften, mit denen sich auch Algorithmen, die dieselbe Aufgabe bewältigen, unterscheiden und für den Einsatz unter unterschiedlichen Bedingungen empfehlen können.

Diese Eigenschaften sollten nicht nur die in dieser Arbeit besprochenen Algorithmen besitzen, sondern sind für die meisten Sensornetzwerkalgorithmen relevant.

Die wünschenswerten Eigenschaften sind:

- **Energieeffizienz:** Wie in 2.2.2 auf Seite 6 bereits beschrieben, bedeutet Energieeffizienz vor allem sparsamer Umgang mit Nachrichten. Um deren Anzahl zu senken, kann ein erheblicher Rechenaufwand betrieben werden ohne die Energiebilanz negativ zu belasten.

- **Speichereffizienz:** Die geringe Größe vorhanden Datenspeichers (4 kB bei der Zielplattform) hat großen Einfluss auf den Entwurf von Algorithmen. Einerseits macht die feste obere Grenze einige Dinge schlicht unmöglich, wie etwa das Speichern eines gesamten Codeabbilds im RAM, andererseits müssen alle Komponenten extrem sparsam sein, um ein größeres System überhaupt zu ermöglichen. Hier kommt auch ein Vorteil von TinyCubus zum tragen: Die Möglichkeit Komponenten nur bei Bedarf zu installieren reduziert den Speicherverbrauch.
- **Zeiteffizienz:** Auch die Dauer der Codeverteilung und der Rollenzuordnung kann relevant sein, hat bei Sensornetzanwendungen aber häufig nur eine untergeordnete Bedeutung.

4.4.1 Konkurrenz dieser Eigenschaften

Leider ist es oftmals so, dass nicht alle dieser Eigenschaften optimal erfüllt werden können. Wie in der Evaluation zu sehen kann beispielsweise auf Kosten von Energie die Zeit verringert werden. Auch in diesem Bereich zeigt sich wieder die Stärke des TinyCubus Frameworks, welches erlaubt die gewünschten Optimierungsparameter zu spezifizieren und dann den passenden Algorithmus auswählt.

4.5 Grundsätze des Softwareengineering

Modularität, Kapselung, Wartbarkeit und Portierbarkeit gehören zu den erstrebenswertesten Eigenschaften von Software. Während Modularität auch sehr gut durch nesC, die verwendete Programmiersprache, unterstützt wird, müssen bei der Kapselung und besonders bei der Portierbarkeit unter Umständen Einschränkungen aufgrund der Ressourcenknappheit von Sensorknoten gemacht werden. Ein Beispiel dafür ist die Nutzung des Tiny Cross Layer Frameworks, mit welchem aber gleichzeitig die negativen Auswirkungen eingeschränkter Kapselung minimiert werden sollen.

4.6 Zusammenfassung

In diesem Kapitel wurden die Anforderungen an die beiden zentralen Algorithmen erläutert. Das nächste Kapitel behandelt das Subsystem Tiny Configuration Engine, in das die Algorithmen eingebettet sind, bevor die Algorithmen selbst beschrieben werden.

5. Tiny Configuration Engine

5.1 Überblick

Dieses Kapitel behandelt die Details der Tiny Configuration Engine, die wie im Kapitel 3 auf Seite 15 beschrieben in das TinyCubus-Framework eingebettet ist. Die zentralen Algorithmen zur Rollenzuordnung und zur Codeverteilung werden in den anschließenden Kapiteln behandelt. Zu dieser Diplomarbeit gehören aber auch der Entwurf der Architektur der Tiny Configuration Engine sowie das Design und die Implementierung der zusätzlich zu den Algorithmen benötigten Komponenten. Diese beiden Punkte werden in diesem Kapitel erläutert.

5.2 Architektur

Bevor auf die Gesamtarchitektur der Tiny Configuration Engine eingegangen wird, soll noch die Aufspaltung der Funktionalität des Codeverteilungsalgorithmus erläutert werden:

- Der 1-Hop-Übertragungsalgorithmus ermöglicht die Übermittlung eines Codeabbilds bzw. einer beliebigen Datenmenge, zwischen benachbarten Knoten (deshalb 1-hop).
- Der übergeordnete Kontrollalgorithmus ist für die Verteilung der Informationen über neue oder jüngere Versionen vorhandener Komponenten zuständig. Er nutzt bei Bedarf die Dienste der 1-Hop-Übertragung.

Auf der Abbildung 5.1 auf der nächsten Seite ist die Architektur der Tiny Configuration Engine abgebildet. Es werden die folgenden Komponenten unterschieden:

- **RoleAssignment** ist die Komponenten, die den Rollenzuordnungsalgorithmus implementiert.

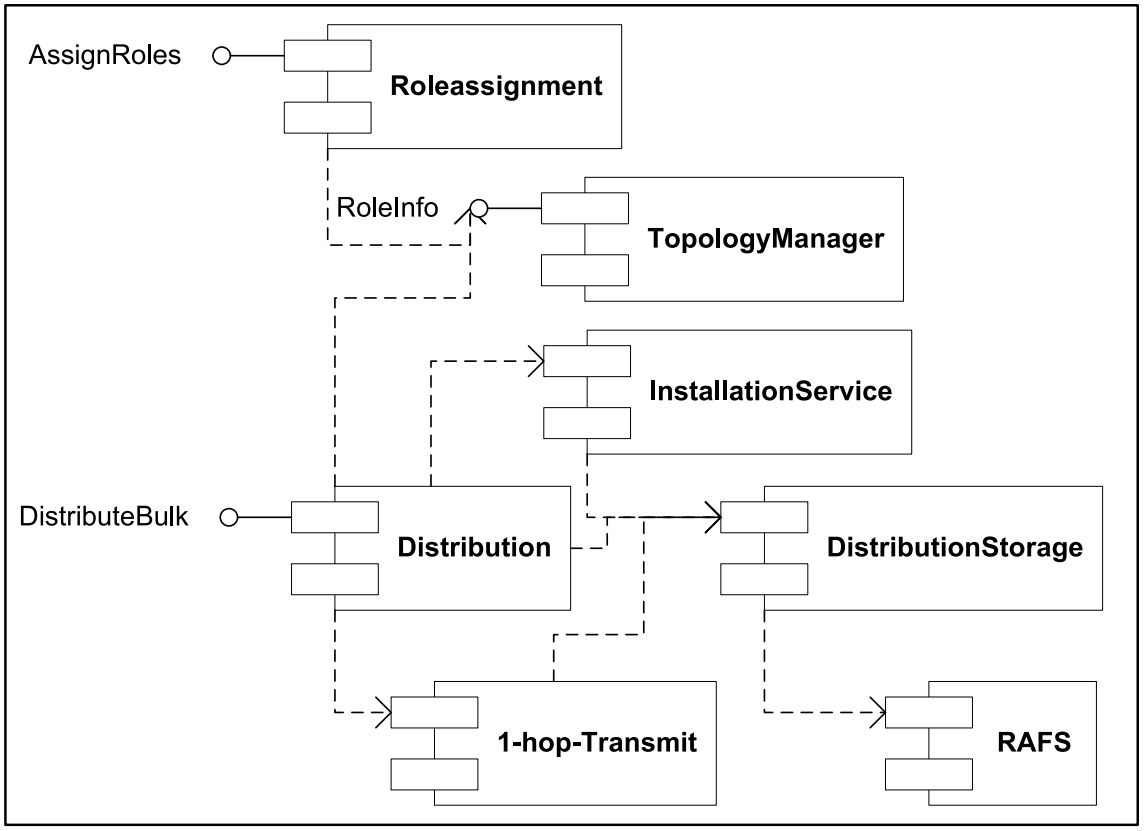

Abbildung 5.1: Architektur des Gesamtsystems

- **Distribution** ist der oben genannte übergeordnete Kontrollalgorithmus.
- **TopologyManager** ist in der jetzigen Implementierung eine einfache, auf die Speicherung von Rolleninformationen der Nachbarschaft eines Knoten spezialisierte, Version des Topologiemanagers.
- **InstallationService** ist die in dieser Diplomarbeit nicht bearbeitete Komponente, die die eigentliche Installation der Komponenten auf den Motes erledigt.
- **1-hop-Transmit** ist die Komponente, die für die eigentliche Übertragung des Codeabbilds zwischen benachbarten Knoten zuständig ist.
- **DistributionStorage** dient zur Speicherung der Codeabbilder inkl. deren Metadaten, wie etwa Prüfsummen.
- **RAFS**, das „Random Access File System“, ist der DistributionStorage Komponente noch nachgeschaltet und bietet eine Dateiabstraktion auf den Flash der Motes.

Diese Komponenten, mit Ausnahme der beiden zentralen Algorithmen, denen separate Kapitel gewidmet sind, werden im Folgenden beschrieben.

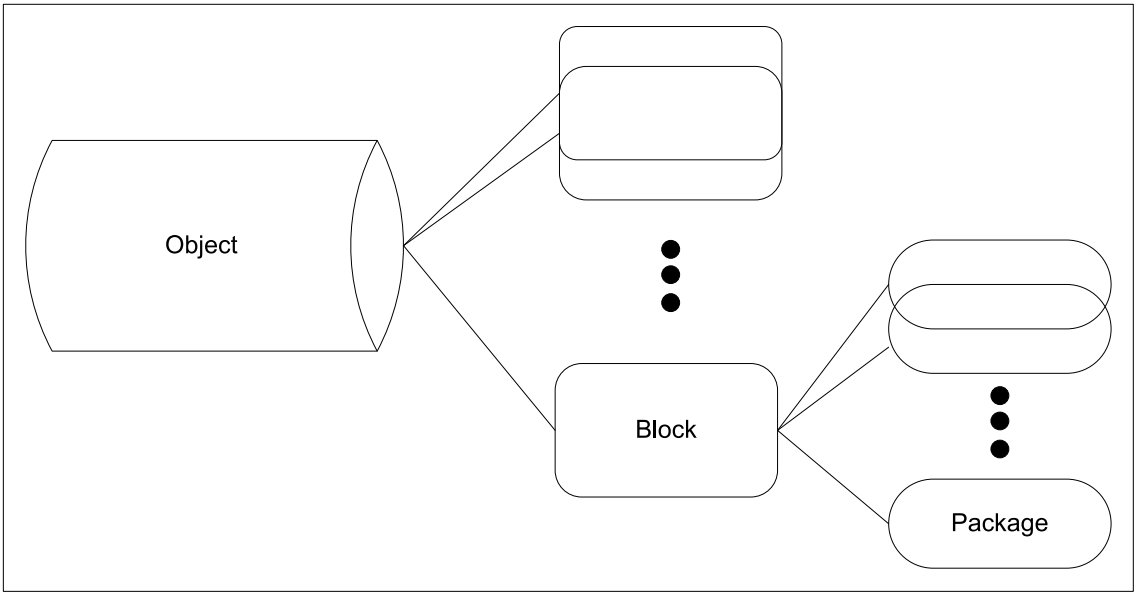

Abbildung 5.2: Hierarchische Aufteilung der Daten für die 1-Hop-Übertragung

5.3 1-Hop-Übertragung (1-hop-Transmit)

5.3.1 Anforderungen an den 1-Hop-Übertragungsalgorithmus

Durch die Trennung zwischen dem übergeordneten rollenbasierten Codeverteilungsalgorithmus und dem Algorithmus für die Übertragung zwischen benachbarten Knoten, sind einige in Abschnitt 4.3 auf Seite 23 Anforderungen durch die 1-Hop-Übertragungskomponente zu erfüllen:

- **Integrität der Daten:** Da dieser Algorithmus die eigentlichen Daten überträgt, ist er auch für die Sicherstellung von deren Integrität zuständig.
- **Fortsetzung von Teilübertragungen:** Die Unterstützung von Teilübertragungen fällt in etwa zu gleichen Teilen in die Verantwortungsbereiche beider Algorithmen.
- **Unterstützung mehrere Komponenten:** Diese Anforderung liegt hauptsächlich im Verantwortungsbereich des übergeordneten Algorithmus. Der 1-Hop Algorithmus muss aber in der Lage sein, Datenströme zu unterscheiden.
- **Robustheit:** Robustheit ist für beide Algorithmen wichtig.

5.3.2 Hierarchische Aufteilung der Daten

Wie in Abbildung 5.2 zu sehen ist, werden die zu übertragenden Daten hierarchisch aufgeteilt. Während das „Objekt“ die Gesamtheit aller Daten repräsentiert, ist das „Package“ die eigentliche Übertragungseinheit. Die Zwischenstufe „Block“ erlaubt es fehlende Pakete effizient als Bitliste codiert (plus Blocknummer) anzufordern.

5.3.3 1-n-Übertragung

Ein zentraler Mechanismus zur Verringerung der gesendeten Nachrichten ist die Nutzung von Broadcasts. Der übergeordnete Verteilungsalgorithmus ist für Aktivierung mehrere Empfänger zuständig, die dann alle von einer Quelle bedient werden können.

5.3.4 Sitzungen

Zur Unterscheidung mehrere Komponenten aus Sicht des 1-Hop-Übertragungsmechanismus werden Sitzungskennungen, die u.a. aus der Adresse des Quellknoten gewonnen werden, benutzt. Auch die Mitteilung dieser Kennung an die Empfänger fällt in die Zuständigkeit des übergeordneten Algorithmus.

Zu diesem Problemfeld gehört auch die Initialisierung der gemeinsam benutzten DistributionStorage (s. Abschnitt 5.5 auf Seite 36) Komponente, etwa das Öffnen der für diese Übertragung zuständigen Datei.

5.3.5 Datennachrichten

Die Datennachrichten sind das Herz des 1-Hop-Übertragungsalgorithmus. Mit ihnen wird jeweils ein Package versendet. Da bei einer maximalen Payload-Größe von 29 Bytes mit hunderten Nachrichten pro Objekt und pro Knoten gerechnet werden muss, hat eine effiziente Ausnutzung dieses Platzes hohe Priorität. Neben den Daten werden deshalb nur Blocknummer, Packagenummer und Sitzungskennung verwendet. Die geschickte Wahl deren Größen hat deshalb spürbaren Einfluss auf die Gesamtleistung des Algorithmus.

5.3.6 Kontrollnachrichten

Da die Hauptlast der Steuerung auf den übergeordneten Algorithmus fällt, benötigt die 1-Hop-Übertragung nur eine Anforderungsnachricht, die fehlende Pakete beim Sender zu deren erneuter Übertragung markiert. Das Verfahren basiert also auf NACKs.

Mit diesem Nachrichtentyp ist auch die Fortsetzung von Teilübertragungen realisiert.

5.3.7 Timeouts

Timeouts dienen zur Robustheit des Algorithmus. Falls beispielsweise der Quellknoten während der Übertragung ausfällt, terminiert der Algorithmus nach einer gewissen Zeit. Die Übertragung kann dann später, auch von einer anderen Quelle, wieder aufgenommen werden.

5.3.8 Prüfsummen

Prüfsummen dienen zur Sicherstellung der Integrität der Daten. Diese sind ebenso wie die Daten hierarchisch angeordnet:

- Pakete erhalten bereits durch TinyOS eine Prüfsumme und werden noch vor der Benachrichtigung der Anwendung durch den Netzwerkstack überprüft und bei Bedarf verworfen. Zusätzlich finden noch Plausibilitätsüberprüfungen statt:
  - Die Sitzungskennung muss der erwarteten entsprechen (ansonsten werden die Pakete ohnehin ignoriert, da sie Teil einer anderen Sitzung sein sollten)
  - Block und Packagenummer müssen im für dieses Objekt gültigen Bereich sein
- Jeder Block erhält eine Prüfsumme. Somit muss, falls doch ein Paket falsche Daten unbemerkt übermittelt hat, nicht das gesamte Objekt, sondern nur ein Block verworfen und erneut übertragen werden.
- Mit einer weiteren Prüfsumme für das gesamte Objekt wird die Integrität der Daten mit sehr hoher Sicherheit gewährleistet.

5.3.9 Kein Pipelining

Pipelining im Sinne von sofortigem Weiterleiten von Paketen wird nicht unterstützt. Sollte dabei die 1-n-Übertragung angewendet werden, muss mit dem Broadcast Strom Problem ([TNCS02]) gerechnet werden. Die Empfänger eines Pakets stören sich gegenseitig, falls mehrere das Paket sofort weitersenden wollen. Dieses Problem wurde auch während der Evaluation (s. 8.3 auf Seite 61) festgestellt.

Eine reduzierte Art des Pipelining ist durch die Aufteilung in Blöcke möglich. Sobald ein Block vollständig und laut Prüfsumme korrekt empfangen worden ist, kann ein Knoten zur Quelle werden. Sollte der Kommunikationskanal zum Sender temporär ausfallen, bevor die Übertragung vollständig ist, kann wenigstens ein Teil der Gesamtdaten bereits weitergegeben werden. Dies kann beispielsweise nützlich sein, wenn die 1-Hop-Übertragung als Teilprotokoll zur Übermittlung einer großen Menge an Sensordaten zu einer Analysestation benutzt wird. Dann kann am Zielknoten die Analyse der bisherigen Daten schon beginnen, während die restlichen Blöcke nach Ende der Störung nachgeliefert werden.

5.4 Dateisystem (RAFS)

5.4.1 Überblick

Da Codeabbilder ein Vielfaches der Größe des Datenspeichers der Motes besitzen können und das direkte Beschreiben des Programmspeichers unmöglich ist,

bietet sich eigentlich nur die Verwendung des Serial Flashs der Motes an. Glücklicherweise stellt TinyOS bereits eine Systemkomponente zu diesem Zweck bereit.

Die Notwendigkeit der Unterstützung mehrere Komponenten lässt dabei eine dateisystemartige Abstraktion am sinnvollsten erscheinen.

Im Rest dieses Abschnitts werden zuerst die Anforderungen an ein solches Dateisystem formuliert. Danach folgt eine Bewertung der bereits verfügbaren Komponenten. Da diese den Anforderungen leider nicht genüge leisten, wurde eine eigene Komponente – das Random Access File System (RAFS) – entworfen, dessen Konzepte und Architektur im Anschluss erläutert werden.

5.4.2 Anforderungen

Im Folgenden sind die Anforderungen an das Dateisystem zusammengefasst:

- **Unterstützung mehrerer Dateien:** Diese Anforderung ergibt sich direkt aus der Unterstützung mehrere Komponenten.
- **Erzeugen von Dateien:** Da nicht alle Komponenten von vorneherein bekannt sind, ja nicht einmal existieren, muss es möglich sein Dateien zu erzeugen.
- **Löschen von Dateien:** Mit dem Update einer Komponente wird ihr altes Abbild überflüssig und sollte aufgrund der Ressourcenknappheit der Sensorknoten gelöscht werden.
- **Durchsuchen der Liste vorhandener Dateien:** Dies ist nötig, um zu erkennen, welche Komponenten bereits vorhanden sind.
- **Wahlfreier Zugriff:** Auf die Daten einer Datei muss wahlfrei zugegriffen werden können um Teilübertragungen zu ermöglichen.
- **Schonung von Ressourcen:** Wie bei jeder Komponente für Sensor Knoten zu beachten.

Hier sind noch die wichtigsten Dinge aufgelistet, die *nicht* benötigt werden, obwohl sie bei einem Dateisystem normalerweise vorhanden sind.

- **Keine nachträgliche Größenänderung von Dateien:** Die Größe eines Codeabbilds ist von vorneherein bekannt und kann beim Erzeugen der Datei angegeben werden.
- **Kein hierarchisches Dateisystem:** Ein hierarchisches Dateisystem ist mit zusätzlicher Komplexität verbunden, deren Aufwand im Vergleich zum Nutzen zu hoch ist. Es ist schließlich nicht mit hunderten oder tausenden Dateien zu rechnen.

- **Sicherheitsfunktionen:** Da es keine Benutzer im herkömmlichen Sinne gibt, sind auch keine Sicherheitsfunktionen notwendig.
- **Koordinierung von Zugriffen mehrere Clients auf dieselbe Datei:** Da in der Regel eine Datei nur von einer Komponente benötigt wird, wird auch kein gemeinsamer Zugriff gestattet, um Ressource zu schonen und die Komplexität des Dateisystems zu verringern.
- **Große Anzahl offener Dateien**: Bei der Ressourcenknappheit wird von einer kleinen Anzahl gleichzeitig aktiver Anwendungen auszugehen, von denen nur ein Bruchteil die Dienste des Dateisystems benötigt. Deshalb ist nur eine kleine Anzahl gleichzeitig geöffneter Dateien vorgesehen, um Ressourcen zu schonen.

5.4.3 Verfügbare Implementierungen

Mit dem „Matchbox filing system“ ([Gay03b] und [Gay03a]) ist bereits eine Implementierung eines Dateisystems für Motes in TinyOS integriert. Daneben besitzt auch Deluge eine Persistenzkomponente.

Die Designkriterien für Matchbox unterscheiden sich in einigen zentralen Punkten von den Anforderungen, die für die Codeverteilung relevant sind. Das übergeordnete Ziel von Matchbox ist die Unterstützung von Logging, etwa dem kontinuierlichen Aufzeichnungen von Messerergebnissen. Matchbox bietet deshalb neben den von einem Dateisystem erwarteten Standardfunktionen, Erzeugen/Löschen/Suchen von Dateien, noch folgende Möglichkeiten:

- **Vergrößerung von Dateien:** Solange Platz im Dateisystem vorhanden ist, können Dateien einfach durch Anhängen von Daten vergrößert werden.
- **Sequentielles Lesen und Schreiben**

Als Ausschlusskriterium erweist sich aber das Fehlen von Funktionen zum wahlfreien Zugriff auf die Dateien.

Die Persistenzkomponente von Deluge, einer TinyOS-Komponente zur Codeverteilung (s. Abschnitt 7.2 auf Seite 51), dagegen bietet, da auf Unterstützung von Übertragungen von Codeabbildern spezialisiert, gerade den wahlfreien Zugriff. Dagegen fehlen die meisten anderen geforderten Funktionen:

- Unterstützung mehrerer Dateien
- Erzeugen von Dateien
- Löschen von Dateien

- Durchsuchen der Liste vorhandener Dateien

Besonders die Unterstützung mehrere Komponenten ist in Deluge nicht vorgesehen. Deluge bietet zwar die Unterscheidung mehrerer Codeabbilder, deren Anzahl und Größe müssen aber bereits zur Compilezeit spezifiziert werden.

5.4.4 Konzepte

Das RAFS nutzt folgende Konzepte, die teilweise denen von Matchbox und anderen „großen“ Dateisystem ähneln:

- **Dateiname:** Durch den Dateinamen wird jede Datei eindeutig identifiziert. Im Gegensatz zu „großen“ Dateisystemen hat er eine feste Länge, kann dafür aber beliebige Bytes enthalten, was etwa die direkte Codierung einer Versionsnummer für eine Komponente ermöglicht.
- **Unterteilung in Blöcke:** Der verfügbare Platz im Flashbereich der Motes wird in Blöcke (nicht zu verwechseln mit den Blöcken der 1-Hop-Übertragung) unterteilt. Da der Flash bereits eine Hardwareunterteilung in Pages besitzt, wird auch diese Größe für die Blöcke verwendet.
- **Freelist:** In der Freelist, einer Bitliste, werden die unbelegten Blöcke gespeichert
- **File Used Block List:** Jede Datei besitzt diesen Eintrag, der die durch sie belegten Blöcke definiert. Durch Einsatz einer Lauflängenkodierung (Startblock plus Anzahl Blöcke) sind nur wenige Bytes pro Datei nötig. Die Tatsache, dass die Dateigröße beim Erzeugen der Datei bekannt ist und somit möglichst wenig zusammenhängende Bereiche reserviert werden können, ermöglicht diese speichereffiziente Kodierung.
- **Dateimetadaten:** Diese Daten werden für offene Dateien im RAM vorgehalten. Dazu gehört neben der File Used Block List nur die Dateigröße.
- **Verzeichniseintrag:** Neben den Dateimetadaten enthält ein Verzeichniseintrag noch den Dateinamen.
- **Verzeichnis:** Auch das Verzeichnis selbst ist im Flash gespeichert und wird im Arbeitsspeicher nur durch einen speziellen Eintrag von Dateimetadaten repräsentiert.

5.4.5 Architektur

Auf der Abbildung 5.3 auf der nächsten Seite ist die Architektur des RAFS zu sehen.

Nach Außen hin bietet das Random Access File System zwei Schnittstellen an:

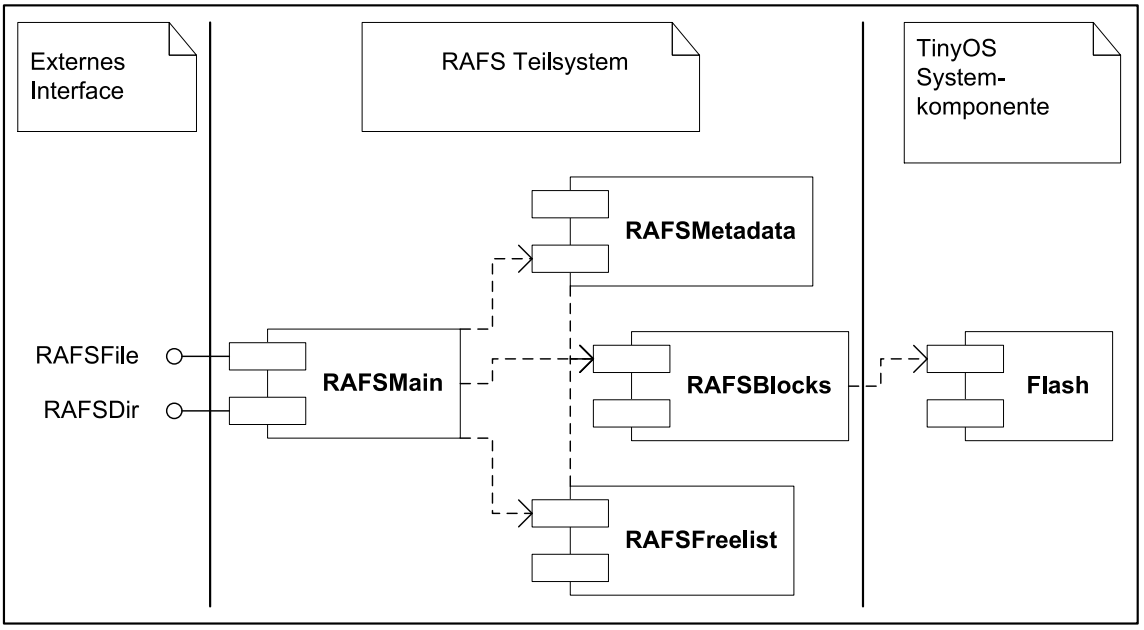

Abbildung 5.3: Architektur des Random Access File Systems (RAFS)

- **RAFSFile:** Dieses Interface bietet Zugriff auf den Dateiinhalt, d.h. Lesen und Schreiben. Daneben sind noch die Operationen zum Öffnen, Schließen und Erzeugen von Dateien definiert.
- **RAFSDir:** Diese Schnittstelle bietet neben dem Durchsuchen der Dateiliste auch die Möglichkeit zum Löschen einer Datei an.

Alle Funktionen sind dabei als Split-Phase-Operationen definiert, da sie in der Regel Flashzugriffe benutzen, die ebenfalls in zwei Phasen ablaufen. Für die Komponenten des RAFS hat dies eine Implementierung als Zustandsautomaten zur Folge.

Vom TinyOS-System wird nur die Flashkomponente benutzt. Alle Zugriffe laufen dabei über das RAFSBlocks-Modul. Die Funktionen der Freelist sowie das Verzeichnismanagement sind in zwei eigenen Komponenten implementiert. Auf diese und RAFSBlocks greift das Hauptmodul zu.

5.4.6 Ausblick

Das Random Access File System ist selbstverständlich nicht nur zur Codeverteilung verwendbar, sondern kann von beliebigen Komponenten benutzt werden. Die Haupteinschränkung ist dabei die fehlende Möglichkeit eine Datei im Nachhinein, am besten auch stetig etwa mit neuen Messwerten, zu vergrößern. Der Verzicht auf diese Funktion hilft aber den Ressourcenbedarf erheblich zu verringern, so dass auch eine etwaige Integration mit Matchbox einer intensiven Untersuchung bedarf.

5.5 Persistenz (DistributionStorage)

Da neben dem eigentlichen Codeabbild auch noch eine ganze Reihe von Metadaten – etwa die Prüfsummen und die Listen bereits empfangener Pakete und Blöcke – gespeichert werden müssen, wurde eine zusätzliche Abstraktionsschicht zwischen den Verteilungsalgorithmen und dem Random Access File System geschaffen. Diese bietet folgende Funktionen und Vorteile:

- **Metadaten und Daten:** Da sowohl der übergeordnete Codeverteilungsalgorithmus als auch die 1-Hop-Übertragung auf Metadaten, teilweise denselben, neben dem eigentlichen Codeabbild angewiesen sein, bietet es sich an diese in derselben Datei zu speichern. Die DistributionStorage Komponente sorgt dabei für die reibungslose Koexistenz.
- **Koordination:** Beide Algorithmen müssen auf die Datei zugreifen. Es bietet sich deshalb an, den Zugriff hier zu zentralisieren und zu koordinieren um die RAFS-Komponente von dieser nicht für jeden Einsatzzweck nötigen Funktionalität zu befreien.
- **Zugriff in der „Sprache“ der Algorithmen:** Diese Zwischenschicht erlaubt den beiden Algorithmen Zugriff auf die Daten mit ihren Begriffen. So ist beispielsweise das Schreiben und Lesen eines „Package“ definiert. Das Umsetzen in Byte-Positionen wird in dieser Komponente erledigt. Das reduziert die Komplexität der Implementierungen der Algorithmen und erhöht die Wartbarkeit des Systems.
- **Gemeinsame Nutzung von Ressourcen:** Durch die Zentralisierung von Ressourcen, wie etwa dem Dateihandle für das RAFS oder RAM-Puffern für zentrale Metadaten, wird der Verbrauch vor allem von Arbeitsspeicher reduziert.

5.6 Topologiemanager (TopologyManager)

In der aktuellen Fassung ist eine noch recht einfache Version des Topologiemanagers implementiert. Neben der Speicherung von Rolleninformationen zur Nachbarschaft, sind das vor allem Analysefunktionen beispielsweise die Anzahl Knoten mit einer bestimmten Rolle für diese lokal gespeicherten Informationen.

Die eigentliche Speicherung der Nachbarschaftsinformationen wird außerdem in Zukunft in das Tiny Cross Layer Framework integriert.

5.7 Zusammenfassung

In diesem Kapitel wurde die Architektur der Tiny Configuration Engine beschrieben. Es folgte eine Erläuterung, der für die zentralen Algorithmen notwendigen Komponenten.

In den beiden nächsten Kapiteln werden nun genau diese Algorithmen behandelt.

6. Generischer Rollenzuordnungsalgorithmus

6.1 Überblick

In diesem Kapitel wird der generische Rollenzuordnungsalgorithmus behandelt. Er bettet sich wie im vorigen Kapitel beschrieben in die Tiny Configuration Engine ein.

Der Algorithmus nutzt die Ideen aus [RFMB04].

Nach einem Überblick über verwandte Arbeiten wird die in [RFMB04] definierte Sprache für Rollenspezifikationen vorgestellt. Es folgt eine Beschreibung des Ablaufs. Danach wird das schon im Kapitel Anforderungen erwähnte Schleifenproblem erläutert. Die Vor- und Nachteile, sowie ein Ausblick beenden dieses Kapitel.

6.2 Verwandte Arbeiten

In [SuH.00] wird u.a. ein Algorithmus zur automatischen Vergabe von eindeutigen Knotenkennungen vorgestellt. Diese Kennungen werden dabei auch für ein Multihop-Routing verwendet. Die automatische Vergabe ist besonders bei großen Netzwerken hilfreich. Da der hier vorgestellte Rollenzuordnungsalgorithmus eindeutige Kennungen der Knoten erwartet, kann der Vergabealgorithmus as Basis benutzt werden.

In [KoSG03] wird ein Algorithmus zu rollenbasierten Organisation von Sensornetzwerken vorgestellt. Er ist allerdings nicht generisch, sondern erzeugt eine vorgegebene Topologie, die von den Autoren als vorteilhaft für eine große Zahl an Sensornetzwerkanwendungen angesehen wird.

6.3 Spezifikationssprache

In [RFMB04] wird eine Sprache für Rollenspezifikationen beschrieben, die hier kurz an einem Beispiel vorgestellt werden soll.

6.3.1 Beispiel

Das folgende Beispiel ist direkt aus [RFMB04] entnommen.

Als Problemstellung wird Definition einer geeigneten Topologie für die Intra-Netzwerk-Aggregation (engl.: In-Network Aggregation) gewählt. Wie in [InGE00] beschrieben ist dies ein Ansatz zu Reduzierung der notwendigen Kommunikation für eine in Sensornetzwerken häufig anzutreffende Aufgabe: das Sammeln der Daten von vielen Knoten und die Übermittlung zu einer Basisstation, welche daraus ein Ergebnis berechnet. Die Idee liegt nun in der Nutzung von Teilberechnungen innerhalb des Netzwerks. Ist beispielsweise der Mittelwert einer Sensorablesung über das gesamte Netzwerk gesucht (etwa die Durchschnittstemperatur), reicht es die Summe und die Anzahl der Messwerte als Ergebnis aus einem Teilnetzwerk weiterzugeben. Durch Nutzung einer hierarchischen Struktur wird der Kommunikationsaufwand gesenkt. Es ist außerdem offensichtlich, dass sich dieses Verfahren für jede Operation anwenden lässt, die sowohl assoziativ als auch kommutativ ist. Es ist aber nicht darauf beschränkt.

Es sind dafür folgende Rollen definiert:

- **Sink:** Das ist die Basisstation. Ihre Rolle erhält sie nicht aufgrund einer Verhandlung, sondern durch ihre herausragende Stellung innerhalb des Netzwerks.

- **Source:** Diese Knoten sind reine Datensammler.

- **Aggregator:** Als Zwischenstationen auf dem Weg der Daten von Datensammlern zur Basisstation, berechnen diese Aggregatoren ein Teilergebnis für ihr Teilnetzwerk und geben dieses weiter. Diese Teilergebnisse können von Aggregatoren einer höheren Stufe wiederum zusammengesetzt werden.

Im Folgenden ist der Quelltext dieser Rollenspezifikation abgebildet.

```
 1:  AGG(0) :: { sensor == temp }
 2:  AGG(N) :: {
 3:  count(2 hops) {
 4:      role == AGG(N-1) &&
 5:      dist(pos, sink_pos) >
 6:        dist(super.pos, sink_pos)
 7:    } >= 2 &&
 8:    count(2 hops) {
 9:      role == AGG(N)
10:    } == 0 &&
11:    1 <= N && N <= 2
12:  }
```

Jede Anweisung besteht aus einer Rolle sowie der Bedingung, die erfüllt werden muss, damit ein Knoten diese Rolle einnehmen kann. Die erste Zeile zeigt ein sehr einfaches Beispiel für eine solche Bedingung: Ein Knoten kann zum Aggregator der Stufe 0, das entspricht einem Datensammler, werden falls er die mit einem Temperatursensor ausgestattet ist.

Die zweite Anweisung zeigt bereits eine Spezialität der Sprache. Es werden mehrere Rollen mit einer Anweisung definiert, indem die Rolle parametrisiert wird. In Zeile 11 wird der bereich des Parameters angegeben, im Beispiel werden also die Rollen AGG(1) und AGG(2) definiert. Dies sind Aggregatoren unterschiedlicher Stufen.

In den Zeilen 8-10 wird ein Beispiel für die „Count“-Funktion gegeben. Diese zählt, innerhalb eines Bereichs (in diesem Fall zwei Hops), alle Knoten die eine Bedingung erfüllen. Im Beispiel zählt die Funktion also alle Aggregatoren, die dieselbe Stufe wie die aktuell untersuchte Rolle besitzen.

In der Count-Funktion der Zeile 3 wird eine weitere Funktion „dist“ eingeführt, die, wie der Name erahnen lässt, die Distanz zweier Knoten berechnet. Außerdem wird über den Präfix „super“ auf eine Eigenschaft des sich in der Auswertung befindlichen Knotens zugegriffen, während innerhalb eine Count-Blocks die Daten der betrachteten (definiert durch den Parameter der Count-Funktion) Knoten verwendet werden.

Diese Spezifikation lässt sich also wie folgt zusammenfassen:

- Wenn ein Knoten einen Temperatursensor besitzt, kann er ein Sammler sein.
- Ein Knoten n wird zum Aggregator einer bestimmten Stufe l, wenn sich in seiner 2-Hop-Nachbarschaft

- Mehr als ein Aggregator der Stufe $l-1$ befindet, und der Knoten n näher an der Basisstation liegt als diese und
- Es noch keinen Aggregator derselben Stufe l gibt.

6.3.2 Relevanz für diese Arbeit

Die Eigenschaften dieser Spezifikationssprache, besonders der Beschreibung der Rollenbedingungen aus Sicht des sich in der Auswertung befindlichen Knotens, wurden als Basis für den Rollenzuordnungsalgorithmus verwendet. Die Spezifikationssprache selbst wird im Moment noch nicht unterstützt. Dies gehört aber zu den Zielen der Weiterentwicklung.

6.4 Annahmen

Es wird angenommen, dass jeder Knoten über eine eindeutige ID verfügt und dieselbe Rollenspezifikation besitzt.

Die Speicherung der Spezifikation kann auf verschiedene Arten erfolgen. Neben einem auf den Knoten laufenden Interpreter, der aber hohe Ressourcenanforderungen stellt, bietet sich die in dieser Arbeit verwendete Möglichkeit einer ausführbaren Komponente an. Die Rollenspezifikation wird dabei zusammen mit dem generischen Algorithmus auf einem normalen Rechner kompiliert und als Komponente auf den Knoten installiert.

6.5 Ablauf

6.5.1 Zustände

Jeder Knoten befindet sich aus Sicht des Rollenzuordnungsalgorithmus in einem der folgenden Zustände:

- **Init:** Dies ist der initiale Zustand, wenn noch keine Rolle an diesen Knoten vergeben wurde. Der Knoten wechselt direkt in den Zustand „Dirty“ und gibt sich die spezielle Rolle „Initial“.
- **InEvaluation:** Wenn ein Knoten gerade die Rollenspezifikation für sich auswertet, befindet er sich in diesem Zustand.
- **Involved:** Alle Knoten, die gerade an einer Auswertung der Rollenspezifikation eines anderen Knoten beteiligt sind, befinden sich in diesem Zustand.
- **Dirty:** Dieser Zustand drückt aus, dass ein anderer Knoten, der für die Rollenauswertung dieses Knoten relevant ist, seine Rolle geändert hat. Dieser Knoten muss also seine Rolle erneut überprüfen.
- **Stable:** Der Knoten besitzt seine Rolle.

Der Zustandsautomat ist in Abbildung 6.1 auf der nächsten Seite zu sehen.

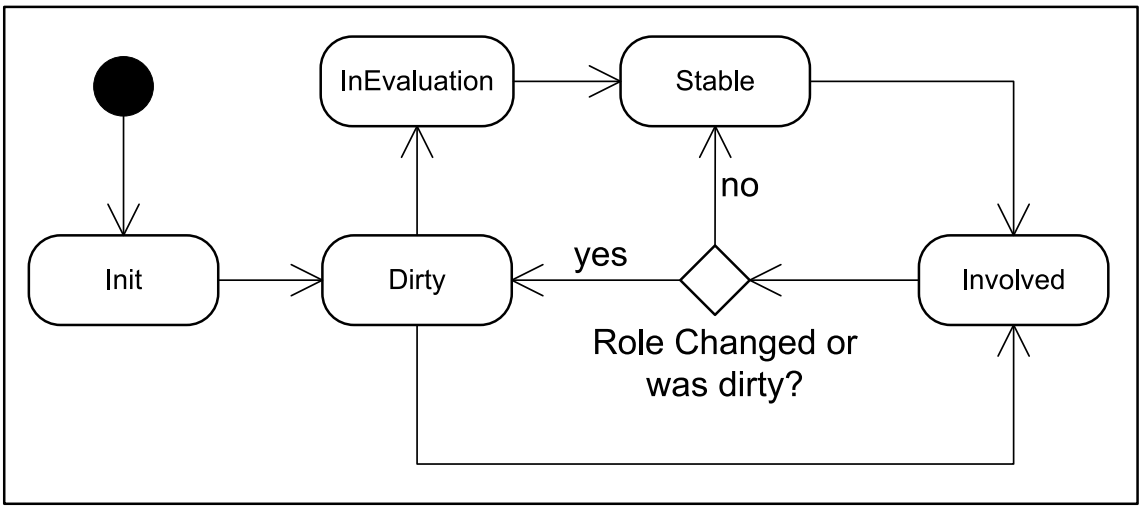

Abbildung 6.1: Zustandsautomat des Rollenzuordnungsalgorithmus (UML Statechart)

6.5.2 Arbeitsweise

Wie in den Anforderungen beschrieben stellt die Nutzung von aktuellen Rolleninformationen der Nachbarn ein Hauptproblem des Algorithmus dar. Wenn ein Knoten sich im Zustand „Dirty“ befindet, d.h. seine Rolle nicht mehr die gemäß der Spezifikation erwünschte sein muss, sendet er eine Anforderung an Knoten bis zu einer maximalen Entfernung. Die Anzahl der Hops, ergibt sich aus der Spezifikation. Befindet sich einer dieser Knoten in einem der Zustände „InEvaluation“ oder „Involved“, d.h. ein anderer Knoten wertet gerade seine Rolle aus, so wird anhand der Mote-ID der beiden konkurrierenden Knoten die Auswertung von einem unterbrochen. Alle Knoten, die die Anforderung erhalten, wechseln in den Zustand „InEvaluation“ und senden die nötigen Daten, die sich ja aus der Rollenspezifikation ergeben, an den sich in der Auswertung befindlichen Knoten. Nachdem dieser alle nötigen Informationen empfangen hat, wertet er seine Rolle aus und sendet das Ergebnis an alle involvierten Knoten. Hat sich seine Rolle geändert wechseln diese Knoten in den Zustand „Dirty“ und führen eine eigene Rollenauswertung durch.

Die Herkunft der für die Auswertung der Rollenspezifikation notwendigen Daten, mit Ausnahme der Rolleninformationen, die vom Topologiemanger verwaltet werden, sind beliebige benutzerdefinierte Funktionen, die beispielsweise Auskunft über Hardwareeigenschaften geben können. Eine Änderung dieser Daten, wird nicht vom Algorithmus selbst überprüft. Die Anwendung muss auf Grund der Änderungen die Entscheidung treffen, ob die Auswertung, nicht nur des aktuellen Knoten sondern auch seiner Nachbarn, erneut nötig ist, und die Komponenten mit dem Rollenzuordnungsalgorithmus entsprechend steuern.

Auch die eigentliche Auswertungsfunktion ist vom Benutzer vorzugeben, da sie die Rollenspezifikation enthält. Diese Prozedur kann auf Analysefunktionen des

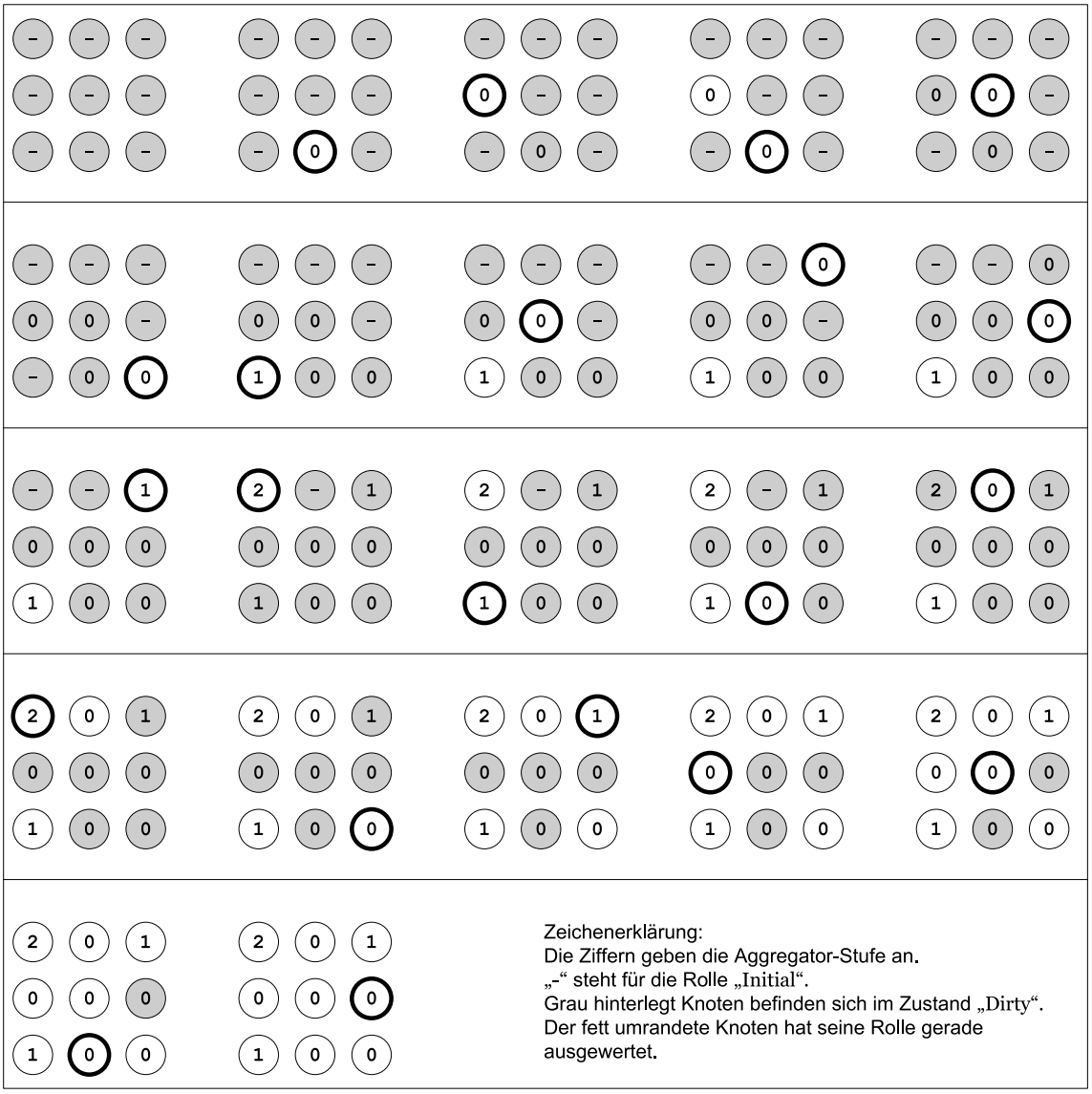

Abbildung 6.2: Beispielablauf für die Intra-Netzwerk-Aggregation

Topologiemanagers, etwa zur Bestimmung der Anzahl von Knoten mit einer bestimmten Rolle in einer n-Hop-Nachbarschaft. Eine manuelle Umsetzung einer Rollenspezifikation, die in der in [RFMB04] definierte Sprache verfasst ist, ist deshalb recht einfach. Ein Codegenerator, welcher diese Umsetzung automatisiert, befindet sich in einer frühen Entwicklungsphase.

6.5.3 Beispiel

In Abbildung 6.2 ist ein Beispielablauf für die Intra-Netzwerk-Aggregation (s. 6.3.1 auf Seite 40) zu sehen. Der Algorithmus läuft auf einer Gitterstruktur mit 3×3 Knoten. Jeder Knoten „hört“ nur seine horizontalen und vertikalen Nachbarn. Die Position der Basisstation wird als links oben befindlich angenommen.

In der resultierenden Topologie befindet sich der Aggregator der höchsten Stufe offensichtlich der Basisstation am nächsten. Die Aggregatoren der Stufe 1 sind

auch offensichtlich mehr als zwei Hops voneinander entfernt, und in der 2-Hop-Nachbarschaft jedes Knoten befindet sich ein Aggregator höherer Stufe. Die geschaffene Topologie entspricht also der durch die Spezifikation vorgegebenen.

6.6 Das Schleifenproblem

Ein Problem eines Rollenzuordnungsalgorithmus ist die mögliche Entstehung von Schleifen und das damit einhergehende Fehlen einer Terminierung.

6.6.1 Beispiel

Das Problem der Entstehung von Schleifen soll an folgendem konkreten Beispiel erläutert werden.

Es sind folgende Rollen definiert:

- Datensammler S
- Koordinator K
- Assistent A

Die Rollen werden nach den folgenden Kriterien aus Sicht jedes Knoten zugeordnet:

- Ein Knoten wird zum Koordinator, wenn
  - In seiner direkten Nachbarschaft mindestens 2 Sammler existieren
  - In seiner direkten Nachbarschaft kein Koordinator existiert
- Ansonsten wird ein Knoten zum Assistenten, wenn
  - In seiner direkten Nachbarschaft mindestens 1 Koordinator existiert
  - In seiner direkten Nachbarschaft kein Assistent existiert
- Ansonsten wird ein Knoten zum Sammler

Einfach ausgedrückt benötigen mehr als ein Sammler einen Koordinator und jeder Koordinator benötigt einen Assistenten.

Auf Abbildung 6.3 auf der nächsten Seite ist nun der Rollenzuordnungsvorgang für eine Netzwerkstruktur mit drei Knoten, die sich alle in Reichweite voneinander aufhalten, aufgezeichnet. Der fett umrandete Knoten hat seine Rolle im jeweiligen Schritt geändert. Zu beachten ist, dass ja nach einer Rollenänderung in dieser Struktur *alle* Knoten ihre Auswertung neu ausführen.

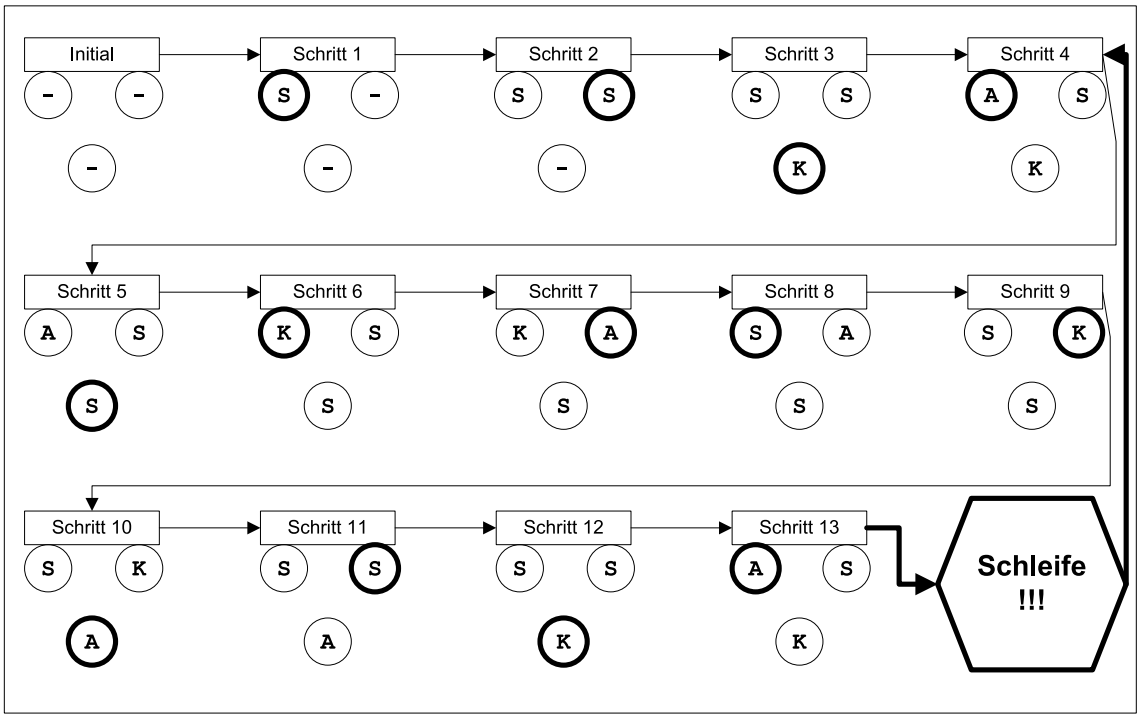

Abbildung 6.3: Beispiel für die Entstehung von Schleifen

Der aufgezeichnete Pfad ist nicht der einzig mögliche. Es ist aber einzusehen, dass alle Wege zu einer Schleife führen, da ein Netzwerk mit drei Knoten keine im Sinne der vorliegenden Rollenspezifikation gültige Topologie einnehmen kann.

Dieses Beispiel zeigt auch sehr deutlich, dass die Existenz von Schleifen nicht nur von der Rollenspezifikation, sondern im besonderen Maße auch von der konkreten Struktur des Netzwerks abhängt. Ein Netzwerk mit vier Knoten führt offensichtlich zu einem stabilen Zustand mit zwei Sammlern und je einem Koordinator und Assistenten.

6.6.2 Lösungsansätze

Es gibt einige Lösungsansätze für dieses Problem:

- **Maximale Anzahl an Rollenänderungen:** Dieses einfach zu implementierende Verfahren erlaubt nur eine (anpassbare) vorgegebene Anzahl an Rollenwechseln pro Knoten. Dieses Verfahren funktioniert garantiert, bei beliebigen Rollenspezifikationen und Netzwerkstrukturen und fordert keine Vorkehrungen des Benutzers. Das Problem ist, dass die resultierende Rollentopologie des Netzwerks nicht notwendigerweise die durch die Rollenspezifikation vorgegebene ist.

- **Keine Rolle erneut einnehmen:** Dieses Verfahren verhindert, dass ein Knoten der eine Rolle bereits zuvor hatte, diese nach einem oder mehreren zwischen zwischenzeitlichen Rollenwechseln erneut einnehmen kann. Die Vor-

und Nachteile sind identisch mit der vorigen Lösung, die Anzahl an Rollenwechseln einzuschränken.

- **Schleifen erkennen:** Eine Schleifenerkennung kann auch zur Terminierung des Verfahrens eingesetzt werden. Da aber dazu das gesamte Netzwerk in Betracht gezogen werden muss, ist ungeklärt, ob und mit welchem Aufwand ein Algorithmus dafür genutzt werden kann.

- **Nur bestimmte Rollenwechsel erlauben:** Dieser Lösungsansatz erweitert die Rollenspezifikation, um nur noch bestimmte Übergänge zwischen den Rollen zu erlauben. Dies erfordert eine Erweiterung der in oben beschriebenen Spezifikationssprache. Die Beschreibung der erlaubten Übergänge kann beispielsweise in Form eines Zustandsautomaten erfolgen. Der Vorteil dieses Verfahrens liegt in der größeren Kontrolle durch den Benutzer, der aber dafür sorgen muss, dass mit der erweiterten Spezifikation Schleifen auch wirklich verhindert werden, was den Hauptnachteil des Verfahrens ausmacht.

- **Schleifenlose Rollenspezifikation:** Dieses Verfahren stellt eine Verallgemeinerung des vorigen Punktes dar. Die Frage, wie die Nichtexistenz von Schleifen bei einer gegebenen Rollenspezifikation bewiesen werden kann, oder wie eine nicht-triviale Spezifikationssprache aufgebaut sein muss, ist von erheblicher Komplexität.

Den ersten drei Verfahren ist gemeinsam, dass die resultierende Topologie des Netzwerks nach der Rollenzuordnung nicht notwendigerweise der durch die Rollenspezifikation erwarteten entspricht. Die auf einer Rollentopologie aufbauenden Algorithmen müssen also auch mit Abweichungen der Topologie arbeiten können.

Die ersten beiden Verfahren sind leicht anzuwenden und zu implementieren. Ein Algorithmus zur Schleifenerkennung oder die Frage, ob eine Spezifikation Schleifen ausschließen kann, gehen aber weit über diese Diplomarbeit hinaus. In diesen Zusammenhang fällt auch das Problem, ob jede gewünschte Topologie überhaupt garantiert ohne die Möglichkeit einer Schleife spezifiziert werden kann.

In der jetzigen Implementierung des Algorithmus wird der erste Ansatz, die Definition einer maximalen Anzahl Rollenänderungen, unterstützt.

6.7 Einschränkungen und Nachteile

Neben dem gerade beschriebenen Schleifenproblem, fehlt auch eine Festestellung der Terminierung des Algorithmus über das gesamte Netzwerk gesehen. Da ein Knoten nicht feststellen kann, ob die Rollenzuordnung in einem anderen Teil des Netzwerks noch aktiv ist, kann er eigentlich nicht in einen Endzustand wechseln.

Je nach Rollenspezifikation und Netzwerkstruktur kann eine einzige Änderung an einem Ende des Netzwerks zu einer komplett veränderten Topologie führen.

Zur Terminierung wird deshalb ein Timeout eingesetzt: Ist ein Knoten eine gewisse Zeit ununterbrochen im Zustand „Stable“, sieht er die Rollenzuordnung als beendet an. Treten danach Rollenänderungen oder –bewertungen in seiner Nachbarschaft auf, informiert der Algorithmus eine benutzerdefinierte Komponente. Diese ist dann für das aus Anwendungssicht korrekte Verhalten, etwa den erneuten Einstieg in die Rollenzuordnung, verantwortlich.

6.8 Vorteile

Der Hauptvorteil der Definition einer rollenbasierten Topologie liegt in der Bereitstellung einer Struktur für andere Algorithmen, die dann effizienter arbeiten können.

Der Vorteil des generischen Rollenzuordnungsalgorithmus, gleichgültig ob vom Benutzer an besondere Bedürfnisse angepasst oder unter Verwendung einer High-Level-Spezifikationssprache, liegt in der reduzierten Komplexität. Damit verringert sich der Aufwand erheblich, der zur Definition einer rollenbasierten Topologie zu erbringen ist.

6.9 Ausblick

Es gibt noch weitere Fragestellungen, die untersucht werden können:

- **Schleifenproblem:** Dieses Problem wird im Abschnitt 6.6 auf Seite 45 ausführlich beschrieben.
- **Terminierungsalgorithmus:** Die Entwicklung eines Algorithmus, der das Ende der Rollenzuordnung im gesamten Netzwerk feststellt und alle Knoten davon informiert, kann das im Abschnitt 6.7 auf der vorherigen Seite beschrieben Terminierungsproblem lösen.

Beiden Punkten ist aber gemeinsam, dass die Fragen nach dem „wie“ oder sogar „ob“ sowie der nötige Aufwand von erheblicher Komplexität sind und weit über diese Arbeit hinausgehen.

Auch eine Umsetzung der Ideen aus [RFMB04] zur parallelen Auswertung von Teilen der Rollenspezifikation auf allen beteiligten Knoten steht noch aus.

Es stellt sich noch die Frage, inwieweit ein anderer Ansatz für eine Spezifikationssprache umgesetzt werden kann. Die in [RFMB04] definierte Sprache beschreibt die Bedingungen für eine Rolle aus Sicht des auswertenden Knotens. Alternativ könnten auch Aussagen über Teilnetzwerke oder das Verhältnis der

Knoten untereinander getroffen werden. Eine Aussage wie „jeder Aggregator einer Stufe mit Ausnahme der höchsten Stufe, soll einen übergeordneten Aggregator in seiner 2-Hop-Nachbarschaft zugehörig sein“ beschreibt die gewünschte Topologie aus einer anderen Perspektive, als es die obige Spezifikationssprache erlaubt. Die Frage inwieweit solche Aussagen formalisiert und vor allem auch ausgewertet werden können (und mit welchem Aufwand), ist noch offen.

6.10 Zusammenfassung

In diesem Kapitel wurde der Rollenzuordnungsalgorithmus beschrieben. Dieser bildet die Basis für den im nächsten Kapitel behandelten rollenbasierte Codeverteilungsalgorithmus.

7. Rollenbasierter Codeverteilungsalgorithmus

7.1 Überblick

Der Codeverteilungsalgorithmus fügt sich wie in den Kapiteln 3 auf Seite 15 und 5 auf Seite 27 beschrieben in das System ein.

Zuerst werden einige verwandte Arbeiten beschrieben. Nach dem Netzwerkmodell und den Annahmen wird auf den Ablauf des Algorithmus eingegangen. Im Anschluss folgen Vor- und Nachteile sowie ein Ausblick.

7.2 Verwandte Arbeiten

Mit XNP ([JeKB03]) ist bereits seit längerer Zeit eine Komponente zur Netzwerkprogrammierung in TinyOS enthalten. Diese unterstützt aber kein Multi-Hop-Routing.

Trickle ([LPCS04]) bezeichnet einen Algorithmus der die Verteilung von Metainformationen von Codeupdates leistet. Trickle ist ein epidemisches Protokoll, das eine Reihe von Techniken zur Unterdrückung unnötiger Mitteilungen über das Vorhandensein con Code-Updates benutzt. Trickle ist für TinyOS implementiert und wird von anderen Arbeiten bereits verwendet.

Deluge ([HuCu04]) ist ein komplettes unter TinyOS laufendes System zur Codeverteilung und -installation. Es baut auf Trickle auf und ergänzt ein Protokoll zur Übermittlung der eigentlichen Daten, sowie die nötigen Komponenten zur Speicherung des Codeabbilds und die Reprogrammierung eines Knoten. Es wird allerdings nur der komplette Austausch des gesamten Systemimages unterstützt. Deluge ist seit kurzem Teil der TinyOS-Distribution und soll XNP ersetzen.

In [StHE03] wird MOAP (Multihop Over-the-Air-Programming), ebenfalls ein epidemisches Protokoll zur Codeverteilung vorgestellt. Es wird dabei u.a. ein Sliding-Window Ansatz untersucht, der möglicherweise auch zu einer Verbesserung für diesen Algorithmus führen kann. Es wird nicht auf die Unterstützung von Komponenten eingegangen und auch Rolleninformationen werden nicht benutzt.

Impala ([LiMa03] und [LSZM04]) ist eine Middleware für Sensornetzwerke. Sie unterstützt auch das Verteilen von Codeupdates. Auf einem Knoten können zwar mehrere Anwendungen installiert sein, von denen aber immer nur eine aktiv ist. Komponenten werden nicht unterstützt.

In [ReLa03] wird ein differenzenbasierter Ansatz zur Reduzierung der übertragenen Datenmengen vorgestellt. Diese Möglichkeit ist komplementär zu diesem rollenbasierten Ansatz.

Mit Maté ([LeCu02]) steht die Implementierung einer Virtual Machine für TinyOS zur Verfügung. Im Gegensatz zur Übertragung binärer Codeabbilder sind damit aber nicht beliebige Komponenten möglich. Auch wird die Übertragung großer Datenmengen nicht unterstützt.

7.3 Netzwerkmodell

Das Netzwerkmodell basiert auf dem von uns in [MLMH+05] definierten:

- Wir unterscheiden zwischen inneren Knoten und Gatewayknoten. Gatewayknoten sind die Quellen für Codeabbilder.
- Jeder Knoten hat eine Rolle.
- Die 1-Hop-Nachbarschaft eines Knoten umfasst alle Knoten, die durch einen Broadcast direkt erreicht werden können.
- Die maximal-k-Hop-Nachbarschaft besteht aus allen Knoten, die mit maximal k Hops erreicht werden können.
- Ein Knoten ist mit einem anderen maximal-k-Hop-Rollen-verbunden für eine Rolle r genau dann, wenn:
  - Beide Knoten dieselbe Rolle r besitzen und
  - es einen Pfad zwischen diesen Knoten gibt und
  - jeder Teilpfad, der nur Knoten enthält, die nicht die Rolle r besitzen, maximal die Länge $k-1$ Hops besitzt
- Ein (Teil-) Netzwerk besitzt Maximal-k-Hop-Rollen-Konnektivität für eine Rolle r genau dann, wenn:

  - Jedes Paar Knoten, bei dem beide die Rolle r besitzen, ist maximal-k-Hop-Rollen-verbunden für die Rolle r

- Für jedes (Teil-) Netzwerk gibt es ein minimales k für eine Rolle r, so dass das (Teil-) Netzwerk Maximal-k-Hop-Rollen-Konnektivität besitzt

7.4 Annahmen

Für den Ablauf des Algorithmus werden mehrere Annahmen getroffen:

- Jeder Knoten besitzt bereits eine Rolle.
- Die Rollen ändern sich während des Ablaufs des Algorithmus nicht.
- Das Codeabbild ist für die Knoten einer Rolle r (der Zielrolle) relevant.
- Jedes Codeabbild besitzt eine eindeutige Kennung.
- Ein geeignetes, bevorzugt minimales, k ist gewählt, so dass das Netzwerk Maximal-k-Hop-Rollen-Konnektivität für die Rolle r besitzt. Dies kann auf mehrere Arten geschehen:
  - Bei einer bekannten statischen Topologie des Netzwerks kann das minimale k errechnet werden.
  - Ergibt sich die Topologie aus einem benutzerdefinierten Rollenvergabealgorithmus, kann ein minimales k eventuell aus dessen Spezifikation abgeleitet werden.
  - Ein geeignetes k kann durch einen Algorithmus aus dem Netzwerk gewonnen werden.
  - Das k kann auf einer, eventuell worst-case, Abschätzung basieren.
- Die Rollen seiner k-Hop Nachbarschaft (für gegebenes k) sind jedem Knoten bekannt.
- Eine neue oder geänderte Komponente wird an einem oder mehreren Gatewayknoten ins Netzwerk injiziert.

7.5 Ablauf

Für ein gegebenes k ist die k-Hop-Version des Algorithmus zu verwenden. Je kleiner k ist, desto effizienter ist der dazugehörige Algorithmus. Die effizienteste Version ist somit der 1-Hop Algorithmus.

7.5.1 Prinzip

In der jetzigen Implementierung baut der Algorithmus auf Flooding auf. Das zentrale Prinzip lässt sich aber auch auf epidemische Protokolle wie etwa Trickle anwenden.

Sobald ein Knoten ein Codeabbild vollständig empfangen hat, wird er zum Sender. Anstatt aber wie bei Flooding die Daten an jeden seiner direkten Nachbarn weiterzuleiten, betrachtet der rollenbasierte Algorithmus die Knoten seiner k-Hop-Nachbarschaft anhand zweier Kriterien:

- Der Knoten besitzt selbst die Zielrolle
- In der $(k-1)$-Hop-Nachbarschaft des Knoten besitzt mindestens ein Knoten die Zielrolle

Nur wenn eines dieser Kriterien erfüllt ist, benutzt der Algorithmus die 1-Hop-Übertragung zur Übermittlung der Daten und erwartet eine Bestätigung des Knoten.

Der Algorithmus terminiert, wenn von allen nach den obigen Kriterien relevanten Knoten eine Bestätigung erhalten wurde, oder ein Timeout (Robustheit) verstrichen ist. Ein Knoten sendet auf Anfrage eine erneute Bestätigung, wenn er das Codeabbild bereits kennt.

7.5.2 Mehrere Gateways

Mehrere Gateways werden durch die Terminierungsbedingungen unterstützt. Die Ausbreitung stoppt, sobald alle relevanten Nachbarn das Codeabbild besitzen

7.5.3 Mehrere Rollen

Der Algorithmus unterstützt auch die Möglichkeit, dass ein Knoten mehrere Rollen besitzt und/oder ein Codeabbild für mehrere Rollen bestimmt ist. Statt einen Knoten auf die Zielrolle selbst zu prüfen, wird dann getestet ob er mindestens eine der Zielrollen besitzt.

7.5.4 Mobile Knoten

Mobile Knoten werden nur begrenzt unterstützt. Sie müssen sich während des Ablaufs des Algorithmus innerhalb der *Maximal-k-Hop-Rollen-Konnektivität* aufhalten um das Codeabbild zu erhalten.

7.6 Einschränkungen und Nachteile

Der Algorithmus besitzt leider auch Einschränkungen und Nachteile:

- **Nachbarschaftsinformationen müssen ausgetauscht werden:** Dies erfordert natürliche zusätzliche Kommunikation. Wir gehen aber davon aus, dass die Informationen auch für andere Algorithmen relevant sind.
- **Zusatzaufwand bei großem k:** Bei großem k steigen der zusätzliche Kommunikationsaufwand und vor allem auch der Speicherverbrauch zur Verwaltung der Nachbarschaftsinformationen stark an. Während der erhöhte Kommunikationsaufwand durch eine Reduktion der 1-Hop-Übertragungen – ein Codeabbild kann schließlich mehrere hundert Nachrichten zur Übermittlung benötigen – noch aufgefangen werden kann, kann das exponentielle Wachstum des Speicherverbrauchs besonders bei dicht besetzten Netzwerken sogar die Grenzen des RAMs sprengen.
- **Bestimmung eines geeigneten k:** Das ist wahrscheinlich das Hauptproblem. Allerdings gehen wir davon aus, dass viele Sensornetzwerke eine bekannte statische Topologie besitzen und sich so für den Einsatz dieses Algorithmus eignen.
- **Geringere Robustheit:** Wenn aufgrund eines Knotenausfalls oder der Störung eines Kommunikationskanal die *Maximal-k-Hop-Rollen-Konnektivität* des Netzwerks verloren geht, können nicht mehr alle Knoten erreicht werden. Es ist aber davon auszugehen, dass bei den meisten Netzwerkstrukturen mehrere, wenn auch weniger als bei Flooding, Knoten ausfallen müssen um diese Problem hervorzurufen.

7.7 Vorteile

Durch die Nutzung von Rolleninformationen besitzt der Algorithmus einige Vorteile:

- **Potentiell geringere Nachrichtenanzahl:** Dies ist das Hauptziel und wird durch die Evaluation bei geeigneten Netzwerkstrukturen auch bestätigt.
- **Potentiell höhere Ausbreitungsgeschwindigkeit:** Da die eigentliche Übertragung eines Codeabbilds erhebliche Zeit benötigt, ist es vorteilhaft wenn auf die Bestätigung weniger Knoten gewartet werden muss.
- **Geringere Belastung „unbeteiligter“ Knoten:** Knoten, für die das Codeabbild nicht bestimmt ist, müssen weder Energie noch Speicher unnötigerweise verschwenden.

Da als Basis Flooding verwendet wird, bietet der Algorithmus noch die Möglichkeit zu einem Wellenverfahren ausgebaut zu werden. Das ermöglicht einerseits eine Rückmeldung, ob der Code wirklich im Netzwerk verteilt worden ist, und erlaubt andererseits auch ein kontrolliertes gemeinsames Umschalten auf eine neue Version einer Komponente.

7.8 Ausblick

Es gibt noch Möglichkeiten, die untersucht werden könnten. Die Komplexität dieser Fragestellungen würde aber wohl jeweils eine eigene Studien- oder Diplomarbeit erfordern:

- **Epidemischer Algorithmus:** Möglicherweise erlaubt die Verwendung eines epidemischen Algorithmus eine weitere Verringerung des Nachrichtenaufkommens.

- **Algorithmus zur Bestimmung des minimalen k:** Ein Algorithmus, der ein minimales k aus dem Netzwerk selbst gewinnt, kann dessen Wahl erleichtern und optimieren. Der Zusatzaufwand muss aber untersucht und gegenüber der gewonnenen Reduktion abgewogen werden.

- **Kontrolliertes Umschalten:** Auch das kontrollierte Umschalten des gesamten Netzwerks auf eine aktualisierte Softwarekomponenten muss gesteuert werden. Eine zu untersuchende Möglichkeit ist, wie oben erwähnt, der Ausbau des Codeverteilungsalgorithmus zu einem Wellenalgorithmus.

- **Adaptivitität:** Eine besonders interessante Weiterentwicklung ist die Integration adaptiven Verhaltens. Ein optimaler Algorithmus würde stets mit einem für den aktuellen Netzwerkteil optimalen k arbeiten. Möglicherweise ist ein lernender Algorithmus, der aus vorangegangen Codeverteilungen Informationen benutzt, ein Schritt in diese Richtung. Damit würde auch die Bestimmung eines geeigneten k an Bedeutung verlieren oder gar entfallen.

- **Installation auf Anfrage:** Die Installation von neuen Komponenten auf Anfrage wie sie das TinyCubus Framework vorsieht, lässt sich nicht mit diesem Algorithmus erreichen. Dazu ist ein Suchprotokoll notwendig. Inwieweit Rolleninformationen zur Optimierung verwendet werden können muss noch untersucht werden. Mit einem solchen Algorithmus lassen sich auch hochmobile Knoten versorgen. Außerdem können auch Knoten, deren Kommunikationskanäle zeitweise gestört waren, so neue Codeabbilder anfordern.

7.9 Zusammenfassung

Der rollenbasierte Codeverteilungsalgorithmus, auf dem der Hauptfokus dieser Arbeit liegt, wurde in diesem Kapitel behandelt. Er basiert auf Flooding und verwendet Rolleninformationen, um über die Weiterleitung von Codeabbildern zu entscheiden. Im nächsten Kapitel folgt nun seine Evaluation.

8. Evaluation

8.1 Überblick

In diesem Kapitel wird die Evaluation des Codeverteilungsalgorithmus dokumentiert. Es wird dabei Tossim zur Simulation verwendet. Zuerst wird der Versuchsaufbau, d.h. die verwendeten Topologien und die gemachten Annahmen, erläutert.

Danach folgt die Evaluation einer vereinfachten Version des Codeverteilungsalgorithmus, die wir für [MLMH+05] vorgenommen haben. Es werden dabei die Daten nicht wirklich übertragen, sondern nur eine einzige Nachricht wird zuverlässig im Netzwerk verteilt. Neben einem ersten Eindruck, wie sich die Nutzung von Rolleninformationen niederschlägt, ist hier vor allem die Vollständigkeit der Netzwerkabdeckung interessant.

Im Anschluss daran wird der gesamte Codeverteilungsalgorithmus evaluiert und seine Leistung mit Deluge verglichen.

8.2 Versuchsaufbau

8.2.1 Netzwerkstruktur

Die Knoten sind gleichmäßig auf einem 4×12 Gitter mit 10 m Abstand angeordnet. Im Folgenden werden die Begriffe Spalten (4) und Reihen (12) benutzt. Als Funkmodell wird ein verlustloses Scheibenmodell mit einem Durchmesser von 15 m benutzt, d.h. jeder innere Knoten besitzt acht direkte Nachbarn. Nachrichten können dabei nur aufgrund von Kollisionen verloren gehen.

Für die Evaluation werden noch abweichende Gittergrößen untersucht. Das Funkmodell ist aber identisch.

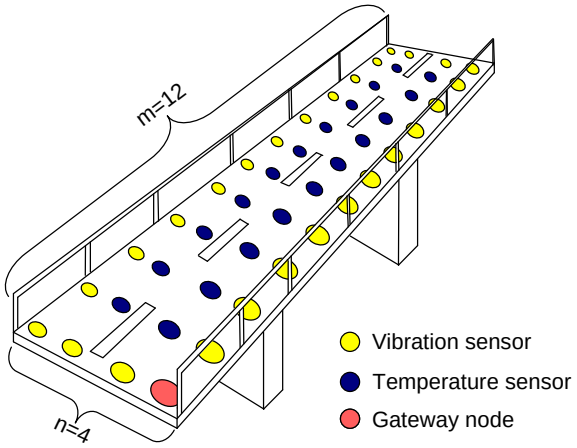

Abbildung 8.1: Die Brückentopologie (aus [MLMH+05])

8.2.2 Topologie

Die Netzwerkstruktur für die Evaluation ist von der Sustainable-Bridges-Anwendung inspiriert. Wie auf Abbildung 8.1 zu sehen, definieren wie zwei Rollen: Vibrations- und Temperatursensoren. Zusätzlich wird noch ein Gateway, das die Codeupdates in das Sensornetzwerk injiziert, definiert. Der Gatewayknoten, identisch mit einem Vibrationssensorknoten, befindet sich dabei in einer Ecke des Gitters.

Die Topologie wird auf zwei Arten variiert, um das Verhältnis der Vibrations- zu den Temperatursensoren zu ändern:

- **Brückenmodell:** Das erste Brückenmodell ist auf der Abbildung 8.1 zu sehen: die Vibrationssensoren sind am Rand, die Temperatursensoren sind in der Mitte. Um das Verhältnis der beiden Rollen zu variieren, wechseln eine zunehmende Anzahl Temperatursensoren ihre Rolle und werden zu Vibrationssensoren. Es wechselt dabei immer eine ganze Reihe ihre Rolle.

- **Zufallsmodell:** In diesem Modell sind die beiden Rollen zufällig entsprechend eines vorgegebenen Verhältnisses verteilt.

8.2.3 Annahmen

Es werden folgende Annahmen gemacht:

- Die Rollenzuordnung ist bereits abgeschlossen

- Es gibt einen Gatewayknoten, in einer Ecke des Gitters, der das Codeabbild ins Netzwerk injiziert.

- Die Ziel der Codeabbilder sind die Vibrationssensoren

Die Rollenzuordnung hat natürlich einen eigenen Verwaltungsaufwand. Da wir aber davon ausgehen, dass eine definierte Topologie von mehreren Anwendungskomponenten genutzt wird, fällt dieser Aufwand nicht allzu sehr ins Gewicht.

Es werden dabei keine echten Codeabbilder übermittelt, da dazu noch keine Daten zur Verfügung stehen. Welche Daten nötig sind, um eine Komponente oder ein Update zu übertragen, wird erst mit Abschluss der Diplomarbeit [Gaug05] feststehen.

8.3 Zuverlässige Übermittlung einer Nachricht

Für [MLMH+05] haben wir untersucht, wie sich ein rollenbasierter Algorithmus im Vergleich zu Flooding bei der zuverlässigen Verteilung einer einzigen Nachricht, im Gegensatz zu einem Codeabbild, im Netzwerk verhält.

Wir haben dabei folgende Messgrößen gewonnen:

- Die Anzahl benötigter Nachrichten
- Die Dauer vom Start des Algorithmus bis der letzte Knoten die Nachricht erhalten hat
- Die Vollständigkeit, d.h. ob alle Zielknoten die Nachricht erhalten haben.

Wir haben, wie oben beschrieben, das Verhalten sowohl bei Brückentopologien als auch bei Zufallstopologien untersucht.

8.3.1 Algorithmus

Um Zuverlässigkeit zu erreichen, werden implizite Acknowledgements benutzt, d.h. ein Sender wartet so lange, bis alle seine Nachbarn die Nachricht weitergeleitet haben. Sollten Nachrichten durch Kollisionen verloren gehen, werden Timeouts aktiv, und den Nachbarn, von denen der Sender noch keine Nachricht empfangen hat, wird eine Bestätigungsanforderung übermittelt.

Da alle Empfänger einer Nachricht zum Sender werden, tritt das Broadcast Storm ([TNCS02]) Problem auf. Um das Problem zu mildern, wartet jeder Knoten eine zufällige Zeit, bis er die Nachricht weiterleitet. Es hat sich gezeigt, dass die Bandbreite dieser Wartezeit erheblichen Einfluss auf die Messergebnisse hat.

Der rollenbasierte Algorithmus entspricht einer einfachen Version des im Kapitel 7 auf Seite 51 beschriebenen. Er baut auf Flooding auf, entscheidet aber anhand von Rolleninformation, ob eine Nachricht weitergeleitet und auf welche Nachbarn gewartet werden muss.

Wir haben die 1-Hop und für zufällige Topologien auch die 2-Hop-Version des Algorithmus untersucht, jeweils mit einer zufälligen Wartezeit in einem Bereich von 150 ms und einem Bereich von 600 ms.

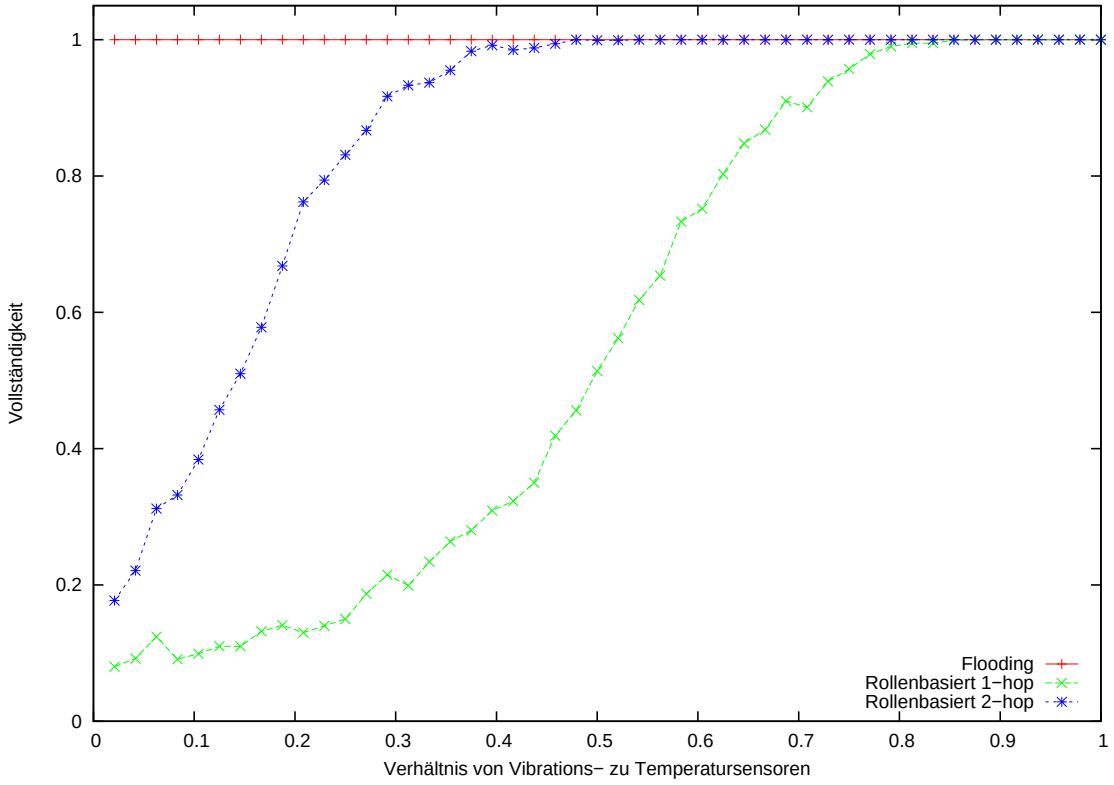

Abbildung 8.2: Durchschnittliche Vollständigkeit bei zufälliger Topologie

8.3.2 Vollständigkeit

Die Vollständigkeit ist nur für die Zufallstopologie interessant. Da in der Brückentopologie für die Rolle Vibrationssensor 1-Hop-Konnektivität besteht, werden immer alle Zielknoten erreicht.

Auf Abbildung 8.2 sind die Ergebnisse zu sehen. Während Flooding natürlich immer alle Knoten erreicht, schafft das die 1-Hop-Version mit sehr hoher Wahrscheinlichkeit erst bei einem Anteil an Vibrationssensoren von 80 %. Die 2-Hop-Variante erreicht alle Knoten mit einer sehr hohen Wahrscheinlichkeit bereits ab einem Anteil von etwa 55 %.

Hier zeigt sich das bereits in Kapitel 7 auf Seite 51 beschriebene Problem der korrekten Wahl der richtigen k-Version des Algorithmus.

Um eine Schätzung von k zu erleichtern wurden für [MLMH+05] zusätzliche Simulationen für größere Werte von k durchgeführt, allerdings wurde dabei nicht der gesamte Algorithmus mit Tossim simuliert, sondern nur sein Verhalten. Auf Abbildung 8.3 auf der nächsten Seite sind die Ergebnisse zu sehen. Für einen gewünschten Grad an Vollständigkeit bei einem gegebenen Rollenanteil kann ein geeignetes k abgelesen werden.

8.3.3 Leistung bei der Brückentopologie

Auf Abbildung 8.4 auf Seite 64 wird die Nachrichtenzahl zwischen dem einfa-

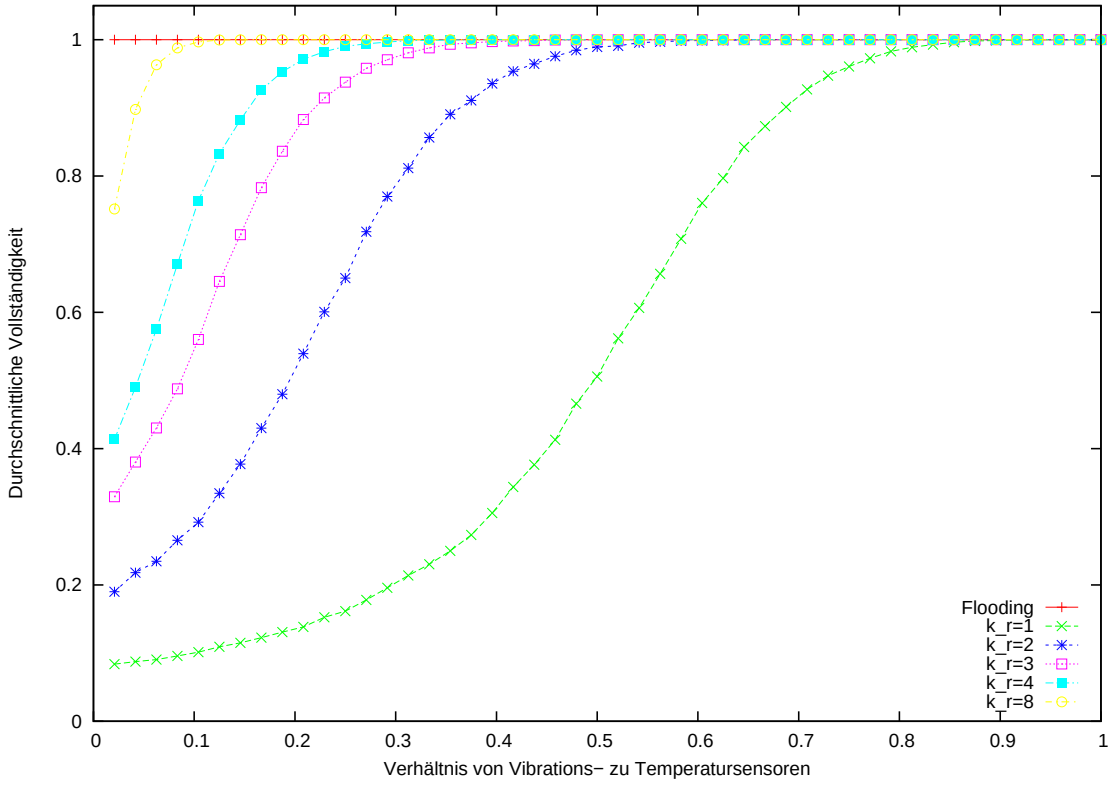

Abbildung 8.3: Durchschnittliche Vollständigkeit bei zufälliger Topologie

chen Floodingalgorithmus und der 1-Hop-Version des rollenbasierten Algorithmus verglichen. Es ist dabei die durchschnittliche Anzahl gesendeter Nachrichten pro Knoten über das Verhältnis von Vibrations- zu Temperatursensoren aufgetragen. Es zeigen sich drei Dinge:

- Die Wahl der Wartezeit hat großen Einfluss auf die Leistung.
- Es werden mehrere Nachrichten per Knoten benötigt. Dies erklärt sich aus der Anforderung nach Zuverlässigkeit.
- Mit beiden Wartezeiten benötigt der rollenbasierte Algorithmus weniger Nachrichten als die entsprechende Floodingvariante.

Es ist außerdem zu erkennen, dass mit zunehmendem Anteil an Vibrationssensoren die Nachrichtenzahl des rollenbasierten Algorithmus steigt. Dies ist zu erwarten, da damit auch die Anzahl der Zielknoten zunimmt. Im Extremfall, wenn nur noch Vibrationssensoren vorhanden sind, verhält sich der rollenbasierte Algorithmus wie Flooding.

Die Dauer vom Start des Algorithmus bis der letzte Zielknoten die Nachricht empfangen hat, ist auf Abbildung 8.5 auf Seite 65 zu sehen. Es zeigt sich, dass Flooding weniger Zeit benötigt, um das ganze Netzwerk zu erreichen. Da der

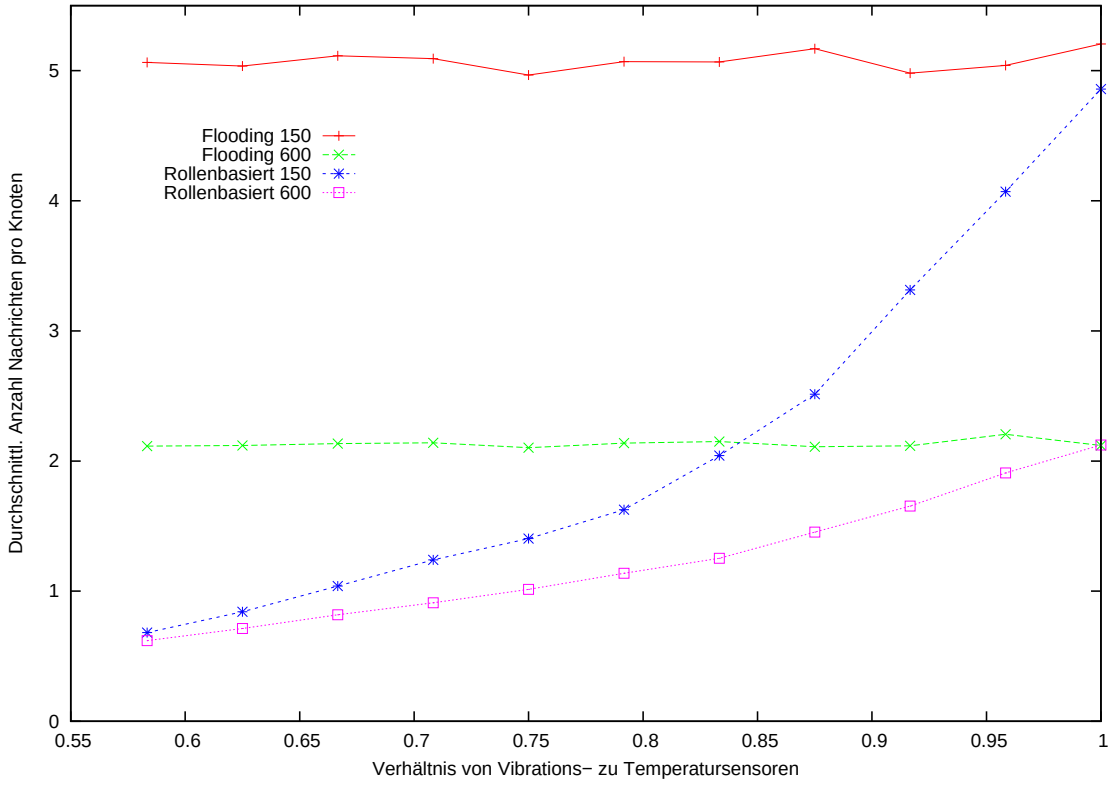

Abbildung 8.4: Durchschnittliche Anzahl gesendeter Nachrichten pro Knoten (Brückentopologie)

rollenbasierte Algorithmus weniger Knoten zur Weiterleitung der Nachricht benutzt, arbeitet er weniger parallel als Flooding. Mit zunehmendem Anteil an Vibrationssensoren nähert sich der rollenbasierte Algorithmus an Flooding an.

8.3.4 Leistung bei zufälliger Topologie

Auf den Abbildungen 8.6 auf Seite 66 und 8.7 auf Seite 66 sind die durchschnittliche Anzahl gesendeter Nachrichten pro Knoten und die Dauer bei einer zufälligen Topologie zu sehen. Diese Messwerte sind nur bedingt aussagekräftig, da die rollenbasierten Algorithmen, wie in Abbildung 8.2 auf Seite 62 zu sehen, nicht notwendigerweise alle Zielknoten erreichen und somit einen unfairen Vorteil besitzen.

Es zeigt sich aber, dass der 2-Hop-Algorithmus, der ja wie oben gezeigt bereits ab einem Anteil von 55 % mit sehr großer Wahrscheinlichkeit alle Zielknoten erreicht, auch in diesem Bereich bedeutend weniger Nachrichten als Flooding benötigt.

8.4 Codeverteilung

In diesem Abschnitt wird nun der komplette Codeverteilungsalgorithmus evaluiert und mit Deluge verglichen. Da es auf mehreren Systemen nicht gelang, Deluge unter Tossim mit mehr als 24 Knoten zu simulieren, wird in diesem Abschnitt

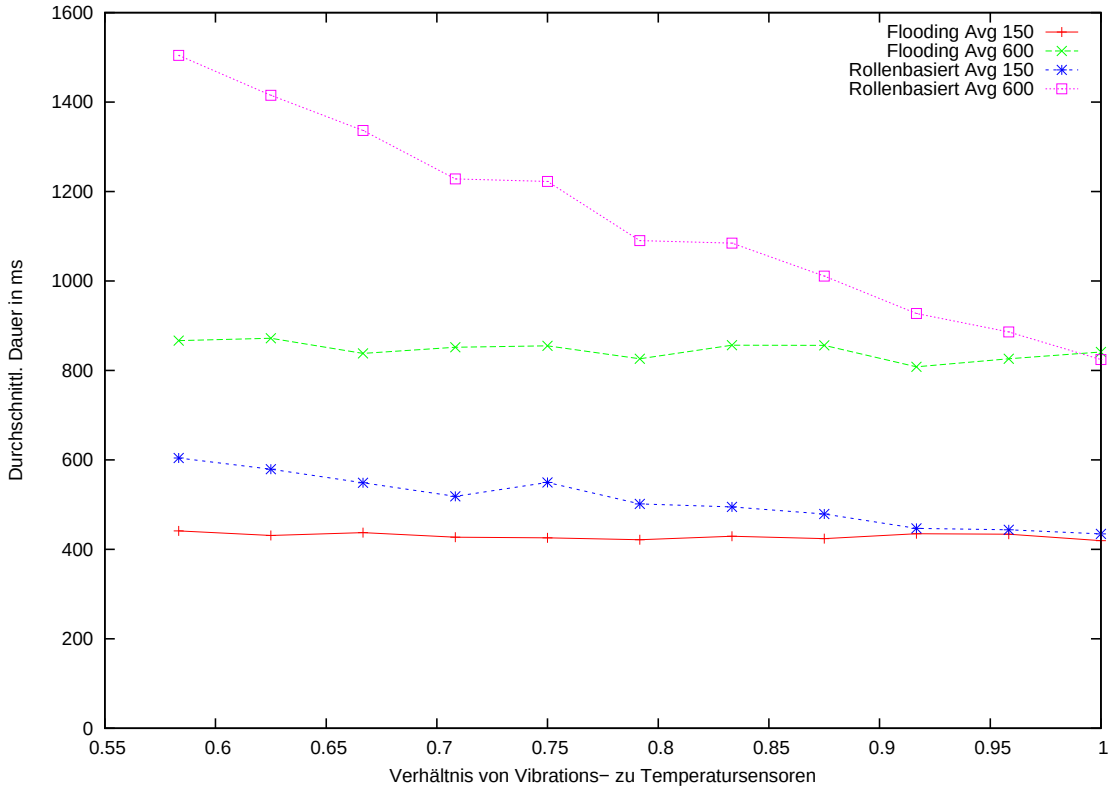

Abbildung 8.5: Durchschnittliche Dauer (Brückentopologie)

nur die halbe Gittergröße mit 4×6 Knoten untersucht. Neben der 1-Hop-Version und der 2-Hop-Version des Codeverteilungsalgorithmus wird auch die Floodingversion untersucht. Diese benutzt zwar ebenfalls Nachbarschaftsinformationen, aber nutzt keine Rolleninformationen um die Nachrichten nur an bestimmte Knoten weiterzuleiten.

Da die Vollständigkeit bereits im vorigen Abschnitt untersucht wurde, wird hier nur die Brückentopologie benutzt.

Wird Deluge für Tossim kompiliert, so wird ein fiktives Codeupdate auf dem Knoten mit der Adresse 0 generiert. Das entspricht genau dem in unserem Experiment benutzten Aufbau. Dieses Codeabbild hat eine Gesamtgröße von 5510 Bytes. Diese Größe wurde auch für die Simulation des rollenbasierten Codeverteilungsalgorithmus benutzt.

Bei der Messung wird zwischen Daten- und Kontrollnachrichten unterschieden. Außerdem wurde bei diesem Experiment die maximale Dauer zwischen dem Beginn des Algorithmus und dem Zeitpunkt, an dem der letzte Knoten das Codeabbild vollständig empfangen hat, gemessen.

Da die Simulationen mit Tossim bei den Algorithmen dieser Komplexität sehr viel Zeit beanspruchen, war es nur möglich eine kleine Anzahl an Testläufen pro Rollenverhältnis durchzuführen. Deshalb schlagen sich größere Abweichungen

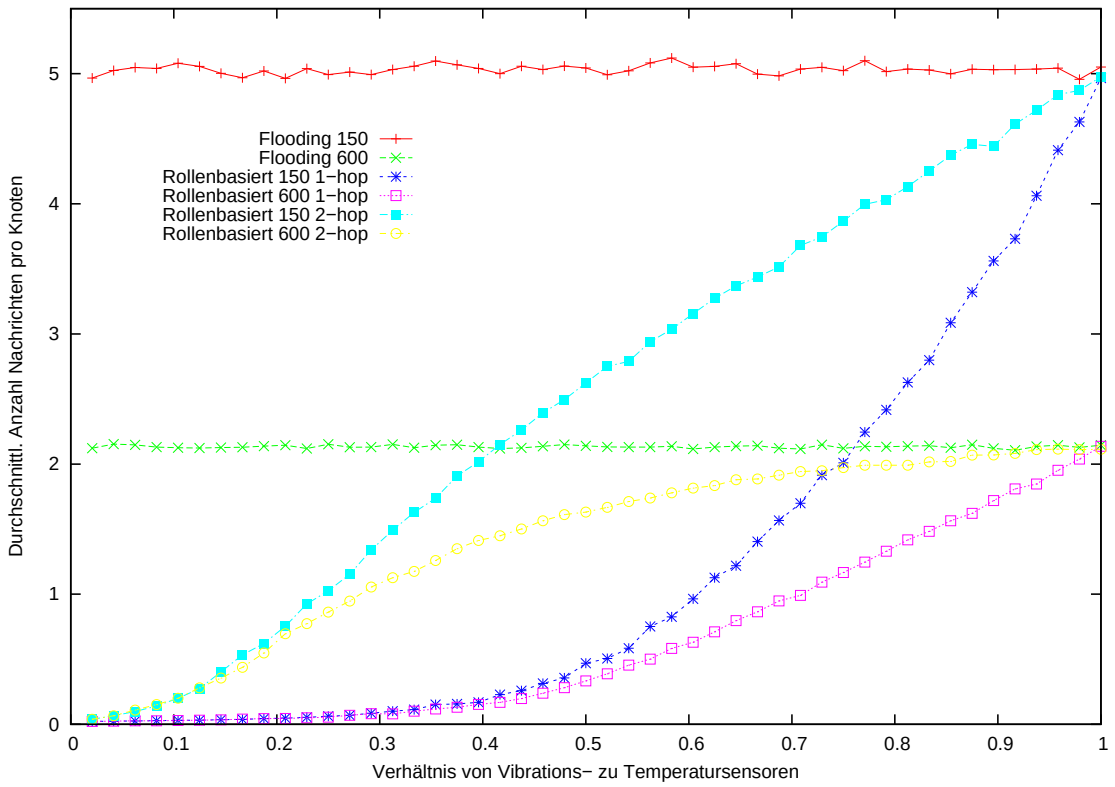

Abbildung 8.6: Durchschnittliche Anzahl gesendeter Nachrichten pro Knoten (zufälliger Topologie)

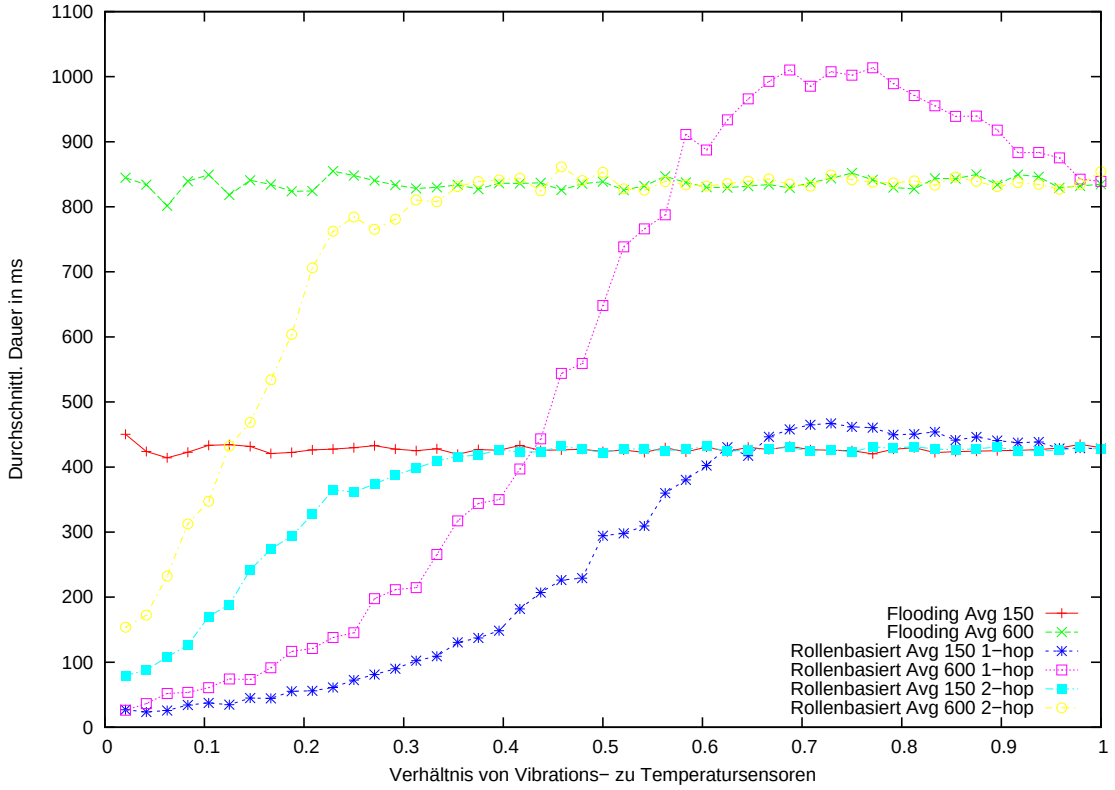

Abbildung 8.7: Dauer bei zufälliger Topologie

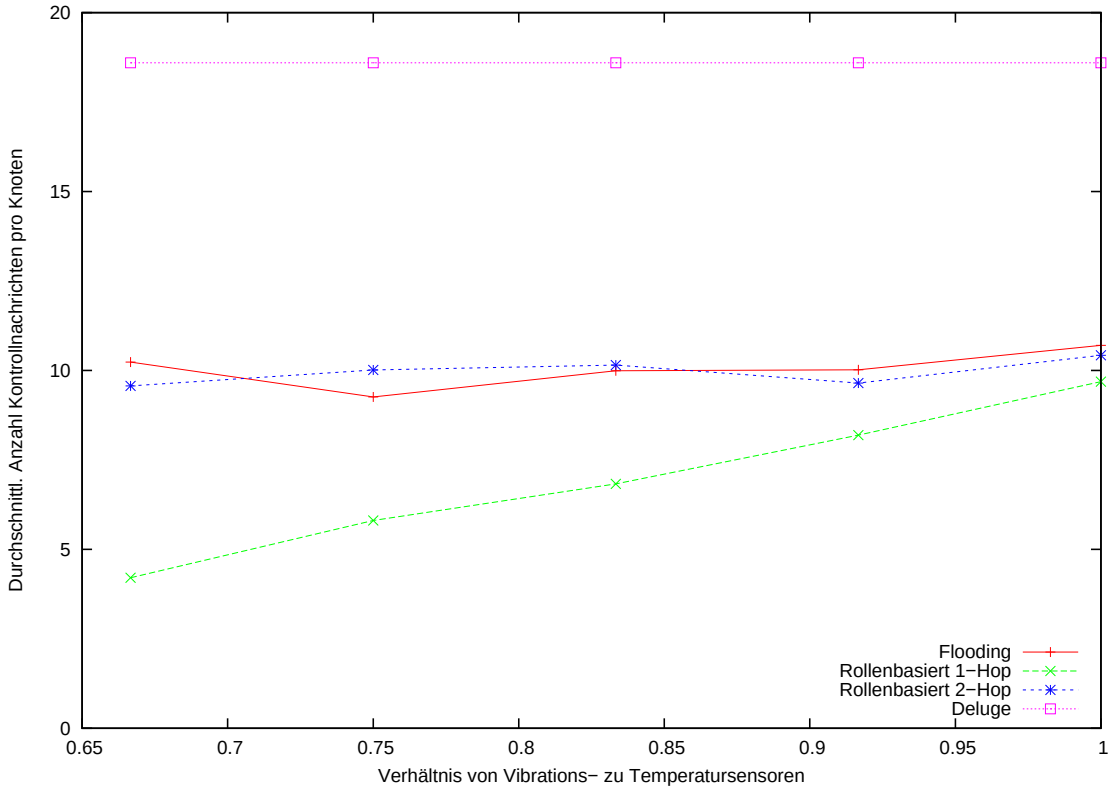

Abbildung 8.8: Durchschnittliche Anzahl gesendeter Kontrollnachrichten pro Knoten

aufgrund von sehr positiven oder sehr negativen Abläufen, stärker nieder, als bei den Experimenten zur Übermittlung einer einzigen Nachricht.

8.4.1 Nachrichtenanzahl

Auf Abbildung 8.8 ist zu erkennen, dass die Anzahl der Kontrollnachrichten, die die 1-Hop-Version des rollenbasierten Codeverteilungsalgorithmus benötigt, bei einer geringen Menge an Zielknoten, bedeutend kleiner ist, als die durch den Flooding oder die 2-Hop-Variante benötigte Anzahl. Dies bestätigt die Erfahrungen mit der Übermittlung einer einzigen Nachricht. Es zeigt sich außerdem, dass schon die Nutzung von Nachbarschaftsinformationen einen Vorsprung gegenüber Deluge ermöglicht, der mit Einbeziehung von Rolleninformationen weiter ausgebaut werden kann.

Die Anzahl der benötigten Datennachrichten ist auf Abbildung 8.9 auf der nächsten Seite zu sehen. Es zeigt sich vor allem, dass die Topologie zu klein ist und die Schwankungen zu groß sind, um klare Vorteile für einen Algorithmus herauszulesen.

Die Nutzung von Broadcasts durch die 1-Hop-Übertragung kann bei einem geringen Anteil an Vibrationssensoren in der gegebenen Topologie zu einem Vorteil für den Floodingalgorithmus werden: Da von inneren Knoten mehr Empfänger gleichzeitig mit Daten versorgt werden können, als von Randknoten. Deshalb ist

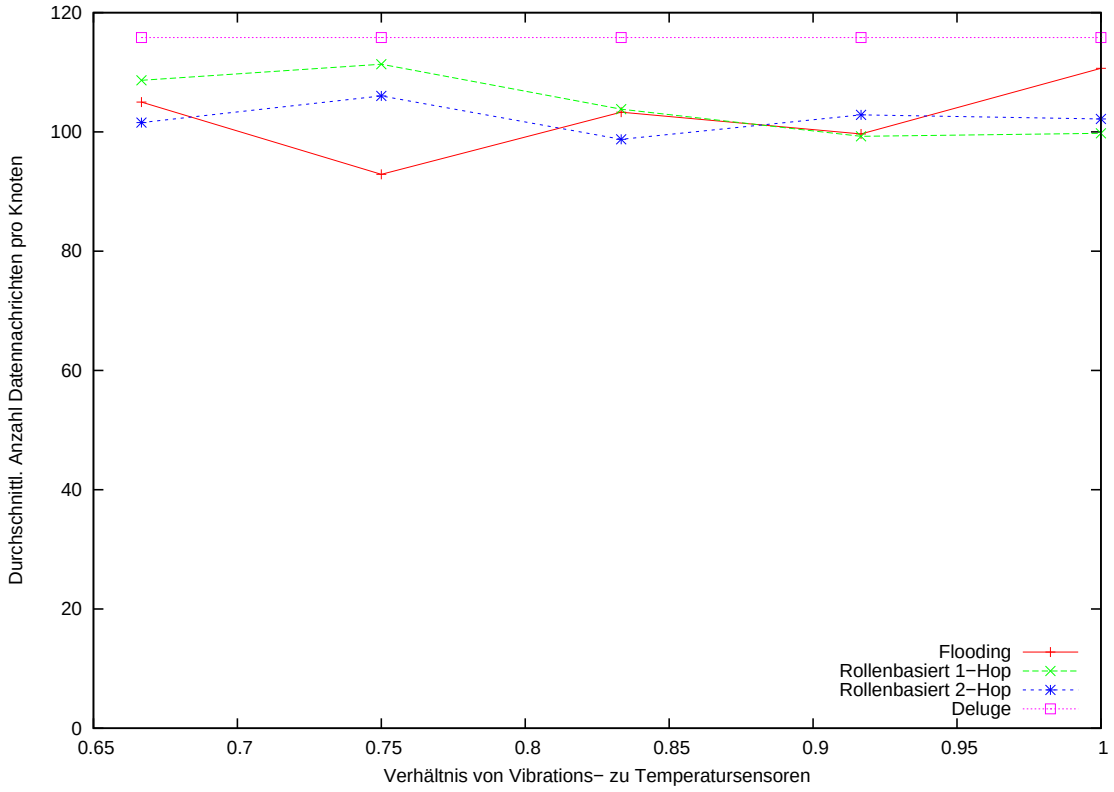

Abbildung 8.9: Durchschnittliche Anzahl gesendeter Datennachrichten pro Knoten

der rollenbasierte Algorithmus im Nachteil, da er bei den Topologien mit einem geringen Anteil an Vibrationssensoren, hauptsächlich oder sogar ausschließlich Randknoten zur Weitergabe nutzt.

8.4.2 Dauer

Bei der Dauer (s. Abbildung 8.10 auf der nächsten Seite) ist der Delugealgorithmus im Vorteil, während die rollenbasierten Algorithmen und der Floodingansatz nahe beieinander liegen, aber etwas abgeschlagen sind. Da auch der Codeverteilungsalgorithmus zufällige Wartezeiten benutzt, um das Broadcast Storm Problem abzumildern, ermöglicht eine Verkürzung des Zeitraums auch eine Verkürzung der Dauer, allerdings auf Kosten eines erhöhten Nachrichtenaufkommens.

8.4.3 Quadratische Struktur

Um das Verhalten der Codeverteilungsalgorithmen anhand einer größeren und breiteren Struktur zu untersuchen wurde abschließend noch ein Experiment mit einem 7×7 großen Gitter durchgeführt.

Auf Abbildung 8.11 auf Seite 70 ist die durchschnittliche Anzahl der Kontrollnachrichten zu erkennen. Auch hier zeigen sich wieder deutlich die Vorteile des rollenbasierten Algorithmus an der Reduzierung der Kontrollnachrichten, und

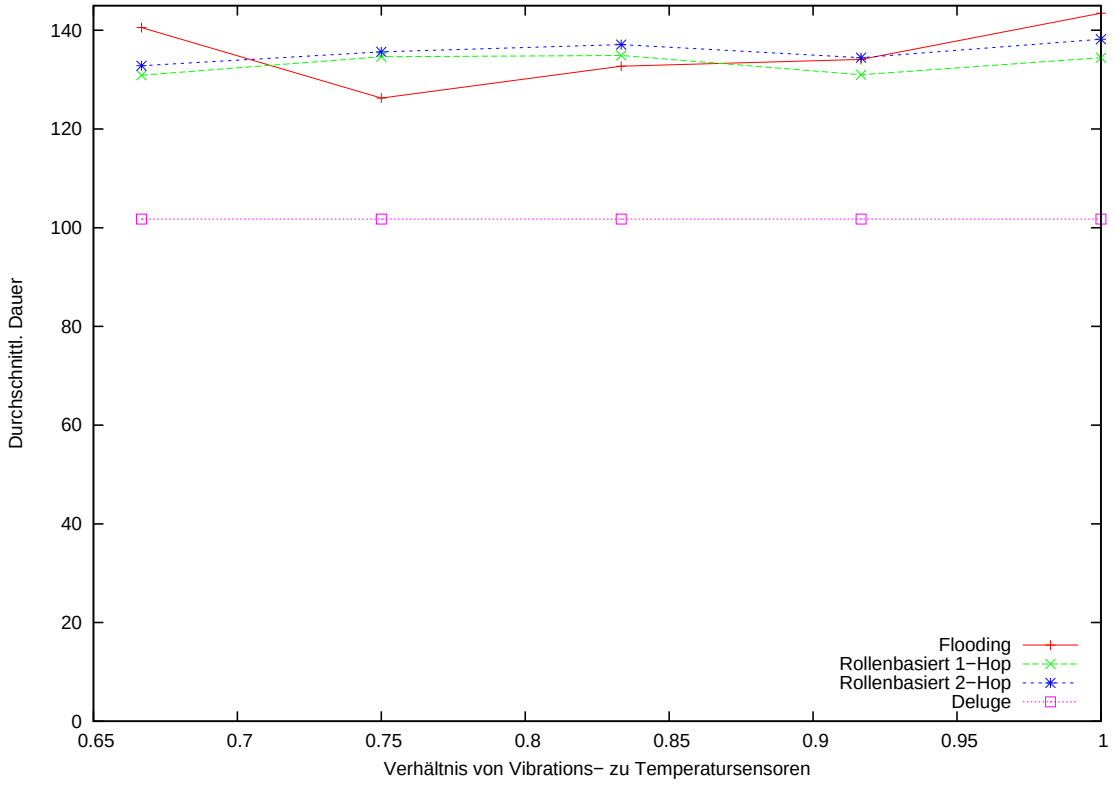

Abbildung 8.10: Dauer der Verteilung im Netzwerk

die Annäherung der Algorithmen bei wachsendem Anteil der Vibrationssensoren.

Auch bei der Anzahl der Datennachrichten (s. Abbildung 8.12 auf Seite 71) zeigt sich jetzt ein ähnliches Verhalten. Allerdings erfolgt die Annäherung aufgrund des Broadcastmechanismus der 1-Hop-Übertragung bedeutend schneller als bei den Kontrollnachrichten.

Die Dauer unterscheidet sich wiederum nicht erheblich zwischen den Algorithmen, wie auf Abbildung 8.13 auf Seite 71 zu erkennen.

8.5 Zusammenfassung

Der rollenbasierte Ansatz birgt im Gegensatz zu Flooding das Problem der mangelnden Vollständigkeit, das eng im Zusammenhang mit der Bestimmung eines geeigneten k, wie im Kapitel 7 auf Seite 51 beschrieben. Die Experimente mit zufälligen Topologien, zeigen aber, dass, wenn eine etwas niedrigere Wahrscheinlichkeit für die Vollständigkeit ausreicht, mit dem 2-Hop-Algorithmus häufig eine gute Wahl getroffen werden kann. Bei einer geeigneten Struktur, wie der Brücke, und einem geringen Anteil an Zielknoten arbeitet der rollenbasierte Algorithmus bedeutend effizienter als ein einfacher Floodingansatz.

Der komplette Codeverteilungsalgorithmus kann bei kleinen Strukturen aus der Nutzung von Rolleninformationen keine großen Vorteile ziehen. Bei großen Struk-

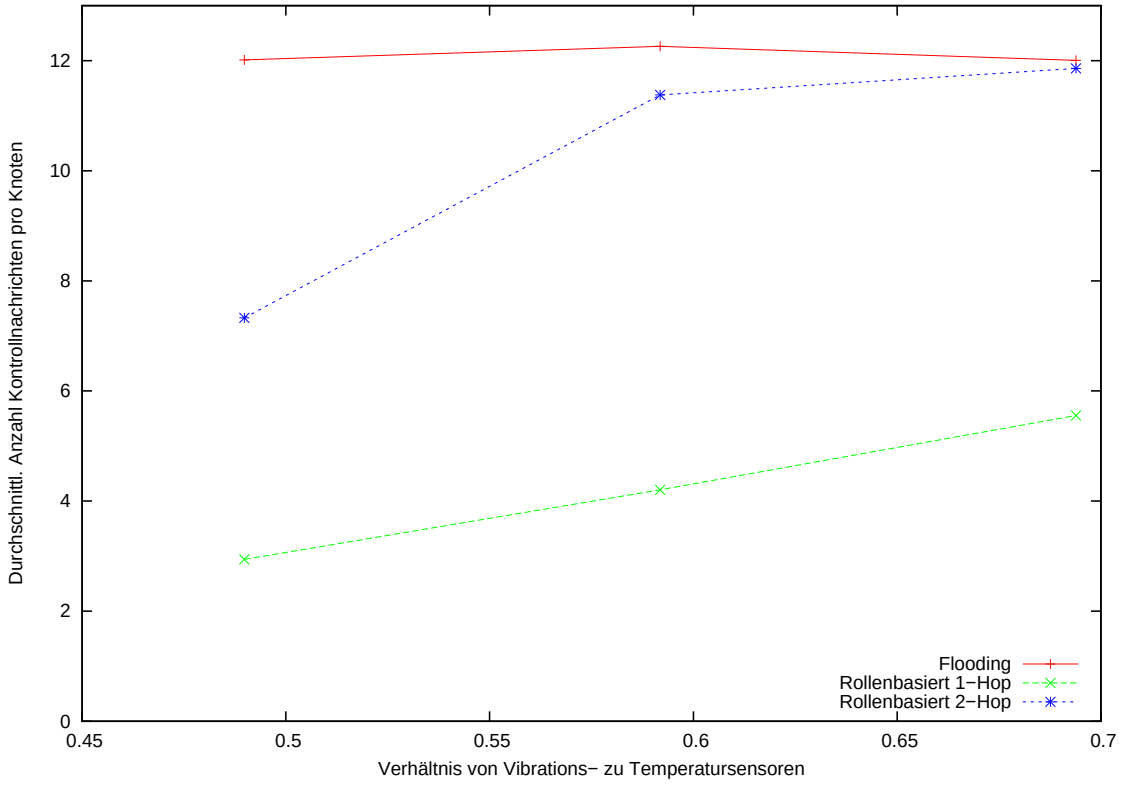

Abbildung 8.11: Durchschnittliche Anzahl gesendeter Kontrollnachrichten pro Knoten (quadratische Struktur)

turen, die wir für die Zukunft vermehrt erwarten, kann der rollenbasierte Codeverteilungsalgorithmus aber bei geeigneten Topologien das Nachrichtenaufkommen erheblich reduzieren.

Nicht untersucht wurden die Vorteile, die sich aus der geringeren Belastung „unbeteiligter“ Knoten ziehen lassen. So erfordert auch das Lesen und vor allem Schreiben des Flashspeichers eine erhebliche Energiemenge. Dadurch, dass Knoten, die eine Komponente nicht benötigen, diese auch nicht weitergeben und damit zwischenspeichern müssen, reduziert sich der Gesamtverbrauch des Netzwerks.

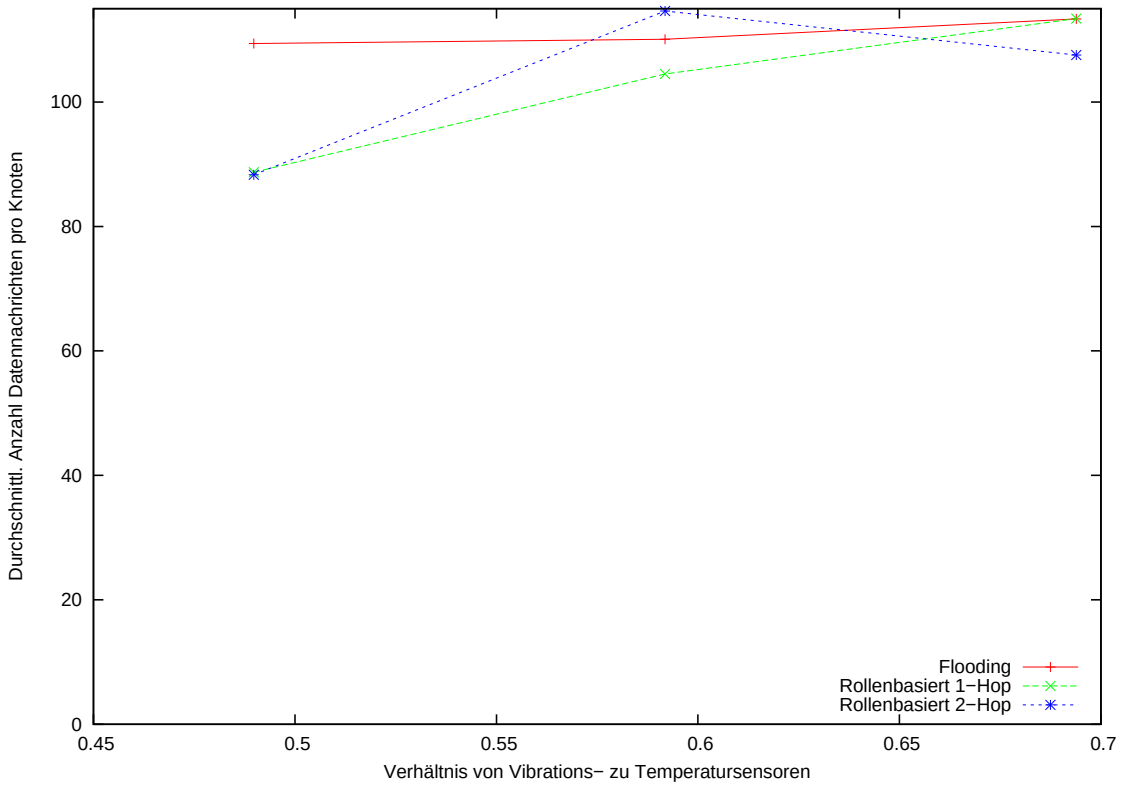

Abbildung 8.12: Durchschnittliche Anzahl gesendeter Datennachrichten pro Knoten (quadratische Struktur)

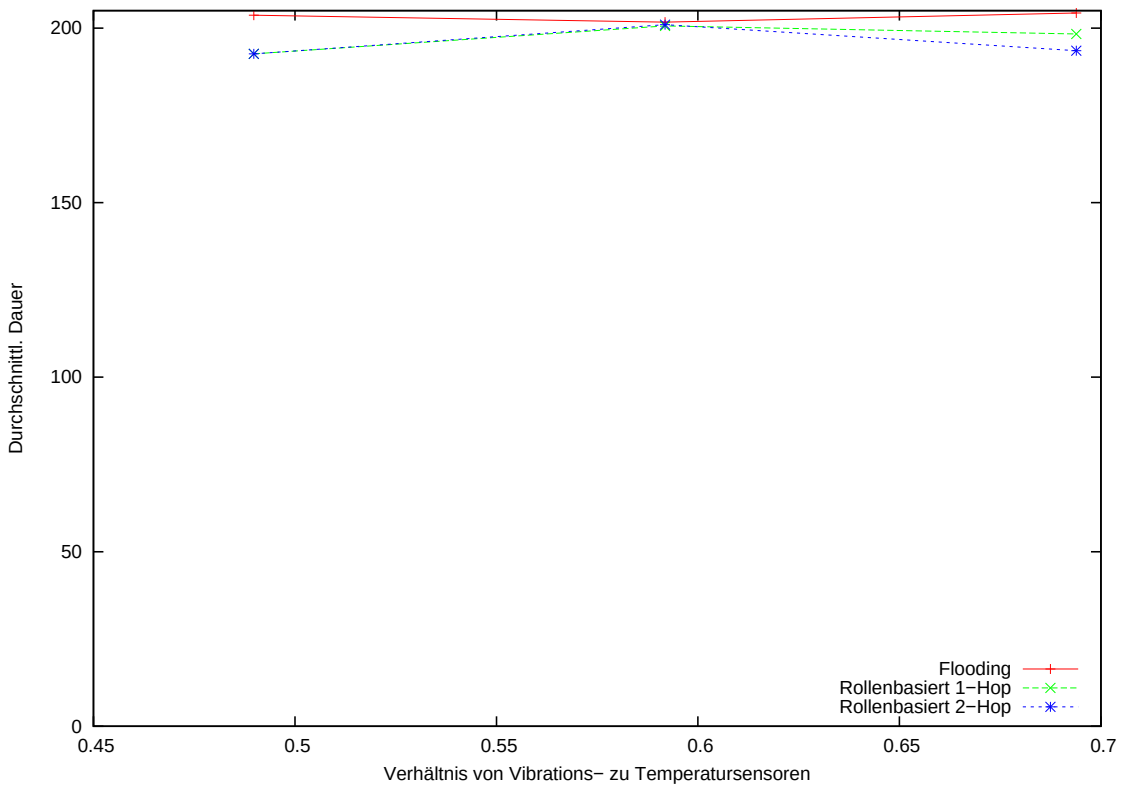

Abbildung 8.13: Durchschnittliche Dauer der Verteilung (quadratische Struktur)

9. Zusammenfassung und Ausblick

9.1 Ausblick

Es sind eine Reihe von Fragestellungen aufgeworfen worden, deren Umfang aufgrund ihrer Komplexität über diese Arbeit hinausgeht.

Für den Rollenzuordnungsalgorithmus steht das ausführlich beschriebene Schleifenproblem an erster Stelle. Ein verwandtes Problem ist die Festestellung der Terminierung des Algorithmus im gesamten Netzwerk.

Für beide Probleme sind pragmatische Lösungen entworfen und implementiert worden. Eine genauere Untersuchung ergibt aber möglicherweise einen effektiveren oder effizienteren Ansatz.

Für den Codeverteilungsalgorithmus ist die Integration von adaptivem Verhalten, d.h. an eine automatische Anpassung an die Netzwerktopologie, ein Ansatz, der möglicherweise eine Reduzierung des Kommunikationsaufkommens ermöglicht. Des weiteren erlaubt ein Algorithmus zur Bestimmung eines optimalen k aus dem Netzwerk heraus, die optimale Wahl der geeigneten Version des Codeverteilungsalgorithmus.

Eine Problem dessen Lösung noch weitestgehend aussteht, ist das Einbeziehen von mobilen Knoten.

Neben diesen umfangreichen Fragestellungen, wurden noch Ideen für konkrete Erweiterungen angedacht. An erster Stelle steht die Integration der eigentlichen Installation von Komponenten, die in der Diplomarbeit [Gaug05] behandelt wird.

Eine Erweiterung des Topologiemanagers zur Verwaltung physischer Koordinaten, erweitert die Optimierungsmöglichkeiten für Algorithmen.

Für den generischen Rollenzuordnungsalgorithmus befindet sich ein Code Generator, der die Spezifikationssprache aus [RFMB04] automatisch umsetzt, bereits

in Entwicklung. Eine parallele Verarbeitung der Rollenauswertung wie ebenfalls [RFMB04] angeregt, erlaubt darüber hinaus eine Reduzierung des Kommunikationsaufwands.

Für den Codeverteilungsalgorithmus sind mehrere Richtungen der Weiterentwicklung denkbar. Einerseits kann die auf Flooding basierende Implementierung zu einem Wellenalgorithmus ausgebaut, und so für ein koordiniertes Aktivieren einer neuen Softwareversion verwendet werden. Andererseits ist die Erweiterung eine epidemischen Algorithmus um die Nutzung von Rolleninformationen vorgesehen.

Ein komplett anderer Algorithmus, ist für die Installation auf Anfrage zu entwerfen. Dieser kann allerdings auf die anderen Komponenten der Tiny Configuration Engine, wie beispielsweise der 1-Hop-Übertragung zurückgreifen.

9.2 Zusammenfassung

In dieser Arbeit wurde ein Großteil der Tiny Configuration Engine, ein Subsystem von TinyCubus, entworfen und implementiert.

Zu den entworfenen und implementierten Komponenten gehören ein Dateisystem und ein Topologiemanager für die Verwaltung von Rolleninformationen. Daneben wurde ein effizienter Mechanismus für die zuverlässige Übertragung von Daten zwischen benachbarten Knoten entwickelt, der auch als Basis für andere Algorithmen dienen kann.

Ein generischer Rollenalgorithmus erlaubt die benutzerdefinierte Definition einer Netzwerktopologie. Diese Topologie kann von anderen Algorithmen zur Optimierung verwendet werden.

Es wurde ein rollenbasierter Codeverteilungsalgorithmus entworfen. Die Evaluation lässt bei größeren Netzwerken und geeigneten Topologien auf eine Reduzierung des Nachrichtenaufkommens gegenüber einem nicht-rollenbasierten Algorithmus schließen. Der Algorithmus erlaubt die zuverlässige Verteilung unterschiedlicher Komponenten im Netzwerk.

Mit Integration der in [Gaug05] entwickelten Komponente entsteht ein System zur dynamischen Installation von Komponenten in Sensornetzwerken.

Literatur

[Corp] Atmel Corporation. **Eine Reihe von Datenblättern und Application Notes,** http://www.atmel.com.

[Gaug05] Matthias Gauger. **Dynamischer Austausch von Komponenten in TinyOS**. Diplomarbeit, Universität Stuttgart,Institut für verteilte und parallele Systeme (IPVS), 2005. In Bearbeitung.

[Gay03a] David Gay. **Design of Matchbox, the simple filing system for motes**, 2003. Teil der TinyOS-Distribution.

[Gay03b] David Gay. **Matchbox: A simple filing system for motes**, 2003. Teil der TinyOS-Distribution.

[GLBW+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer und David Culler. **The nesC language: A holistic approach to networked embedded systems**. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, 2003, S. 1–11.

[HHKK04] Jason Hill, Mike Horton, Ralph Kling und Lakshman Krishnamurthy. **The platforms enabling wireless sensor networks**. *Commun. ACM* 47(6), 2004, S. 41–46.

[HuCu04] Jonathan W. Hui und David Culler. **The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, 2004, S. 81–94.

[InGE00] Chalermek Intanagonwiwat, Ramesh Govindan und Deborah Estrin. **Directed diffusion: a scalable and robust communication paradigm for sensor networks**. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM Press, 2000, S. 56–67.

[JeKB03] Jaein Jeong, Sukun Kim und Alan Broad. **Network Reprogramming**. University of Berkeley, August 2003.

[KoSG03] M. Kochhal, L. Schwiebert und S. Gupta. **Role-based hierarchical self organization for wireless ad hoc sensor networks**. In *WSNA*, San Diego, USA, September 2003.

[LeCu02] Philip Levis und David Culler. **Maté: a tiny virtual machine for sensor networks**. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM Press, 2002, S. 85–95.

[LeLe03] Phlip Levis und Nelson Lee. **TOSSIM: A Simulator for TinyOS Networks**, September 2003. Teil der TinyOS Distribution.

[LiMa03] Ting Liu und Margaret Martonosi. **Impala: A middleware system for managing autonomic, parallel sensor systems**. In *Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2003, S. 107–118.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh und David Culler. **TOSSIM: accurate and scalable simulation of entire TinyOS applications**. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM Press, 2003, S. 126–137.

[LMGP+04] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A. Brewer und David E. Culler. **The Emergence of Networking Abstractions and Techniques in TinyOS.** In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, 2004, S. 1–14.

[LPCS04] Philip Levis, Neil Patel, David E. Culler und Scott Shenker. **Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks**. In *1st Symposium on Networked Systems Design and Implementation (NSDI 2004), March 29-31, 2004, San Francisco, California, USA, Proceedings*, 2004, S. 15–28.

[LSZM04] Ting Liu, Christopher M. Sadler, Pei Zhang und Margaret Martonosi. **Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet**. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004, S. 256–269.

[Meie04] Ulrike Meier. **Fokus Sensoren (Einleitung)**. *Technology Review (dt. Ausgabe)* Band 2, Februar 2004, S. 67.

[MLMH+05] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter und Kurt Rothermel. **TinyCubus: A Flexible and Adaptive Framework for Sensor Networks**. In *Proceedings*

of the 2nd European Workshop on Wireless Sensor Networks, 2005. To appear.

[Raun04] Max Rauner. **Sensible Netze**. *Technology Review (dt. Ausgabe)* Band 2, Februar 2004, S. 68–76.

[ReLa03] Niels Reijers und Koen Langendoen. **Efficient code distribution in wireless sensor networks**. In *Proc. of the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Appl.*, 2003, S. 60–67.

[RFMB04] Kay Römer, Christian Frank, Pedro José Marrón und Christian Becker. **Generic Role Assignment for Wireless Sensor Networks**. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004. S. 7–12.

[RMMM+02] Dirk Reichardt, Maurizio Miglietta, Lino Moretti, Peter Morsink und Wolfgang Schulz. **CarTALK 2000: Safe and Comfortable Driving Based Upon Inter-Vehicle-Communication**. In *Proceedings of the Intelligent Vehicle Symp.*, Band 2, 2002, S. 545–550.

[StHE03] Thanos Stathopoulos, John Heidemann und Deborah Estrin. **A Remote Code Update Mechanism for Wireless Sensor Networks**. Technischer Bericht CENS-TR-30, University of California, L.A., November 2003.

[SuH.00] L. Subramanian und R. H.Katz. **An Architecture for Building Self-Configurable Systems**. In *MobiHoc*, Boston, USA, August 2000.

[Sust] **Sustainable Bridges Homepage** http://www.sustainablebridges.net.

[Tinya] **TinyCubus Homepage** http://www.ipvs.uni-stuttgart.de/abteilungen/vs/forschung/projekte/tinycubus/start.

[Tinyb] **TinyOS Homepage** http://www.tinyos.net.

[TNCS02] Yu-Chee Tseng, Sze-Yao Ni, Yuh-Shyan Chen und Jang-Ping Sheu. **The broadcast storm problem in a mobile ad hoc network**. *Wirel. Netw.* 8(2/3), 2002, S. 153–167.