Aus dem Institut für Telematik
der Universität zu Lübeck

Direktor:
Prof. Dr. rer. nat. Stefan Fischer

Service-orientierte Infrastrukturen und Algorithmen für praxistaugliche Sensornetzanwendungen

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
– Aus der Technisch-Naturwissenschaftlichen Fakultät –

Vorgelegt von
Herrn Dipl.-Inf. Martin Lipphardt
aus Bad Hersfeld

Lübeck, Oktober 2009

Martin Lipphardt
Institut für Telematik
Universität zu Lübeck
Ratzeburger Allee 160
23538 Lübeck
E-Mail: lipphardt@itm.uni-luebeck.de

Dissertation zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
der Technisch-Naturwissenschaftlichen Fakultät
der Universität zu Lübeck

1. Berichterstatter/Berichterstatterin: Prof. Dr.-Ing Christian Werner

2. Berichterstatter/Berichterstatterin: Prof. Dr. rer. nat. Volker Linnemann

Tag der mündlichen Prüfung: 18. Dezember 2009

Zum Druck genehmigt. Lübeck, den 11. Januar 2010
gez. Prof. Dr. rer. nat. Jürgen Prestin
– Dekan der Technisch-Naturwissenschaftlichen Fakultät –

Kurzfassung

Mit dem Fortschritt auf den Gebieten der Miniaturisierung und der drahtlosen Kommunikation hat die drahtlose Sensornetztechnologie immer größere Bedeutung im Bereich der Überwachung (*monitoring*) und Verfolgung (*tracking*) von Phänomenen. Durch die spontane eigenständige Vernetzung der Sensorknoten und die Unabhängigkeit von zusätzlichen technischen Infrastrukturen, findet die Sensornetztechnologie ihre Anwendung in den verschiedensten Bereichen von Militär, Katastrophenschutz, Logistik, Anlagenüberwachung bis hin zur Medizin und Pflege.

Jedes Anwendungsszenario stellt dabei individuelle Anforderungen an das Sensornetz. So unterscheiden sich die Netze durch Knotenanzahl und -dichte, Datenaufkommen, Kommunikation- und Bewegungsmuster. Auch die verfügbare Funktionalität auf den ressourcenarmen Sensorknoten unterscheidet sich in Abhängigkeit von dem Anwendungsszenario. Zur erfolgreichen Entwicklung praxistauglicher Sensornetzanwendungen müssen dabei diese Rahmenbedingungen berücksichtig werden, was spezielles Wissen in den Bereichen der verteilten Systeme und der drahtlosen Sensornetze erfordert.

Die vorliegende Arbeit präsentiert eine Infrastruktur, welche das Erstellen und Anpassen von individuellen Sensornetzanwendungen ermöglicht. Dazu hat der Autor das Paradigma der Service-Orientierung auf Sensornetze übertragen. Die Basiskomponente der Infrastruktur bildet ein neuartiges service-orientiertes Sensorknotenbetriebssystem. Die Grundidee des Systems besteht darin, nur minimale Funktionalität auf den Knoten bei der Ausbringung anzubieten. Sämtliche Funktionalität in den Schichten des ISO-OSI Modells wird in Form von Diensten mittels eines neu entwickelten Codemigrationsverfahren nach der Ausbringung der Knoten hinzugefügt und kann permanent an sich ändernde Bedingungen angepasst werden. Dem Entwickler einer Sensornetzanwendung stellt die Infrastruktur dazu ein Dienstverzeichnis sowie eine grafische Migrationskontrolle zur Verfügung, die es erlauben, verschiedene Anwendungen durch die Auswahl von Diensten zu komponieren.

Der Autor präsentiert zwei Algorithmen, die speziell für Funknetze entwickelt wurden. Beiden Algorithmen liegen dabei die Ideen zu Grunde, dass die *Broadcast*-Eigenschaft des Funkmediums ausgenutzt wird und die Knoten im Netz autark Entscheidungen auf Basis lokal vorhandener Informationen treffen. Der erste Algorithmus platziert dabei selbstorganisierend Dienste so im Sensornetz, dass eine vom Entwickler vorgegebene Abdeckung der Dienste im Netz erfüllt und auch bei Topologieänderungen aufrechterhalten wird. Der zweite Algorithmus realisiert ein Wegewahlverfahren, welches lediglich durch „Lauschen“ am Datenverkehr Informationen über die Netztopologie sammelt und somit Weiterleitungsentscheidungen gänzlich ohne spezielle Wegewahlpakete trifft. Beide

Algorithmen sind als Dienste für das neue Betriebssystem entwickelt worden und in der vorgelegten Arbeit theoretisch und praktisch untersucht.

Zur praktischen Evaluation der neu entwickelten Infrastruktur und Algorithmen wurde im Rahmen dieser Arbeit eine neuartige robuste Sensorknotenplattform entwickelt, die insbesondere über verschiedene Benutzerschnittstellen verfügt, um für verschiedenste praktische Anwendungen unter Nicht-Labor-Bedingungen einsetzbar zu sein.

Die Arbeit legt den Schwerpunkt auf die praktische Anwendbarkeit von Algorithmen und Infrastrukturen für drahtlose Sensornetze. Durch die Übertragung des service-orientierten Paradigmas trägt die Arbeit dazu bei, dass die Technologie der Sensornetze einem größeren Nutzerkreis zugänglich gemacht wird. Die Arbeit stellt einen Schritt zur transparenten Nutzung der Sensornetztechnologie dar.

Abstract

Ongoing progress in the fields of miniaturization and wireless communication has enabled wireless sensor networks to gain importance in the scope of monitoring and tracking phenomena. The autonomic ad-hoc networking of sensor nodes and the resultant independence from additional technical infrastructure has been used to address diverse domains including military applications, disaster management, logistics, industrial plant monitoring, medical applications and healthcare

Every application scenario makes specific demands of a sensor network, resulting broad variation in terms of the number of nodes, node density, data rate, communication and movement patterns. Additionally, the functionality available on resource constrained sensor nodes varies with regards to the application scenario. For the successful development of a real-life sensor network application such preconditions must be considered, necessitating a deep understanding of the fields of distributed systems and wireless sensor networks.

This work presents an infrastructure that allows the development and modification of individual sensor network applications through the application of the service-oriented paradigm. The foundation of this infrastructure is a novel service-oriented sensor node operating system, which minimizes the available functionality of the nodes. All functionality within the layers of the ISO-OSI Reference Model will be added in form of services after the deployment of the nodes using a newly developed code migration algorithm. The service configuration on the nodes can be permanently adapted to changing conditions. The infrastructure provides application developers a service repository and a graphical migration control, which facilitate the composition of various applications through a selection of services.

The author presents two algorithms that were designed specifically for radio networks. Both algorithms exploit the broadcast nature of the radio medium and enable every node in the network to make decisions autonomically based on local knowledge. The first algorithm places services within a network in a self-organizing manner so that a predefined coverage of the service is fulfilled and maintained, even when the network topology changes. The second algorithm is a routing algorithm that gathers information about the network topology by only listening to the traffic on the radio channel. Based on this information, the nodes can make forwarding decisions without using special routing packets. Both algorithms are implemented as services for the new operating system and are analyzed theoretically as well as in practical deployments.

In the scope of this work, the author developed a novel and robust sensor node platform to allow for a practical evaluation of the newly developed infrastructure and algorithms.

Furthermore, the platform provides different user interfaces for use in real-world applications under non-laboratory-conditions.

This work focuses on the applicability of algorithms and infrastructures for wireless sensor networks in real-world deployments. By applying the service-oriented paradigm, this work helps to make sensor network technology accessible for a wider range of users. This work represents a step towards transparent usage of sensor network technology.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Algorithmenverzeichnis

1 Einleitung

Computer finden sich heutztage in allen Lebensbereichen. Neben der Forschung, der Industrie und der Medizin haben sie auch längst den Einzug in unser alltägliches Leben gefunden. Von Mobiltelefonen, die mittlerweile weit über die reine Funktionalität eines Telefons hinausgehen, über Unterhaltungselektronik wie MP3-Player bis hin zum Haushaltsgerät sind Computer fester Bestandteil unseres Alltags. Der amerikanische Wissenschaftler Mark Weiser prägte dafür in seinem Aufsatz *The Computer for the 21st Century* [117] von 1991 den Begriff *Ubiquitious Computing*, die Allgegenwart der Rechner.

Computer finden sich überall dort, wo Daten bzw. Informationen erzeugt, gespeichert und wiedergegeben werden. So speichert ein modernes Mobiltelefon neben Telefonnummern gleichzeitig Adressen und Fotos, verwaltet Termine und kann komplette Historien über Kurzmitteilungsdialoge wiedergeben. Der technologische Fortschritt erlaubt es, immer mehr Technik immer günstiger auf immer kleinerem Raum zu realisieren. Ein MP3-Player speichert etliche Musikalben auf kleinstem Raum und kann diese wiedergeben. So gibt es bereits MP3-Player in der Größe einer Kreditkarte, die 4 GByte Daten speichern können [115]. Mit dieser Entwicklung geht einher, dass die Computer sich immer mehr in ihre Umgebung integrieren und sogar zu verschwinden scheinen.

Parallel zu der Miniaturisierung schreitet auch die Entwicklung im Bereich der *Drahtlosen Kommunikation* voran. Neben dem bereits verbreiteten *Wireless Loca Area Network* (WLAN) und dem *Universal Mobile Telecommunications System* (UMTS), die dem Benutzer erlauben, kabellos per Funk Zugang zum Internet zu bekommen, können sich Geräte auch spontan bzw. *ad hoc* vernetzen und Informationen austauschen. Ein Mobiltelefon beispielsweise verbindet sich heutzutage beim Einsteigen in ein Auto selbstständig mit der Freisprechanlage des Wagens. Diese Vernetzung des Alltags wird als *Pervasive Computing* bezeichnet.

Aus den ständigen Weiterentwicklungen in diesen beiden Bereichen der Miniaturisierung und des Pervasive Computings ergeben sich neue technologische Möglichkeiten. Insbesondere profitiert die Technologie der *Sensornetze* von diesen Entwicklungen. Sensornetze bestehen aus etlichen Kleinstcomputern mit Funkschnittstelle (im Folgenden auch *Sensorknoten* oder *Knoten*), die sich durch spontanes Verbinden untereinander zu einem Netz organisieren und mittels Sensoren ihre Umgebung überwachen. Die Sensorknoten funktionieren völlig autark und verfügen über eine eigene Energiequelle. Die Funkreichweite der Sensorknoten ist jedoch stark begrenzt, um Energie und Platz zu sparen. Die aufgezeichneten Sensordaten werden daher durch das durch die Knoten gebildete Netz an eine Datensenke (beispielsweise ein Rechner am Rande des Netzes) weitergeleitet, von der sie ausgewertet und verarbeitet werden. Somit kann nur das Netz als Gesamt-

heit seine Aufgabe erfüllen. Mittels dieser Technologie können beispielsweise Bereiche überwacht und beobachtet werden, die dem Menschen unzugänglich sind oder in denen der Mensch als Beobachter das zu beobachtende Phänomen beeinflussen würde.

Vorläufer der Sensornetztechnologie war das bereits 1951 ausgebrachte Unterwasserüberwachungssystem *SOSUS* [109] (Sound Surveillance System) der US Navy, welches zur Überwachung sowjetischer U-Boot-Bewegungen diente. Eine SOSUS-Installation besteht dabei aus mehreren Unterwassermikrofonen (Hydrophonen), die über Unterwasserkabel mit einer Überwachungsstation (NAVFACS - *Naval Facilities*) an der Küste verbunden sind, wo die Daten analysiert und ausgewertet werden. Mit dem Fortschritt bei Miniaturisierung und Funkübertragung bekam die Forschung für diese Art der Datenerfassung Ende der 90er Jahre neue Impulse. Insbesondere durch das durch die amerikanische *Defense Advanced Research Projects Agency* (DARPA) geförderte Forschungsprojekt *SMART DUST* [111] der Universität Berkeley wurde die Sensornetztechnologie weltweit zum Gegenstand der Forschung. Ziel des SMART DUST Projektes war es, die für einen Sensorknoten notwendigen Komponenten Sensorik, Funk, Energieversorgung aus bereits verfügbaren Hardwarekomponenten auf kleinstmöglichem Raum zu vereinen. SMART DUST zeigte, dass die notwendige Hardwaretechnologie vorhanden ist, um Sensornetze zu realisieren. Die Forschung breitete sich danach auch auf algorithmische Probleme in Sensornetzen aus. Neben der Hardware werden unter anderem Probleme der Wegewahl innerhalb des Netzes, der eigenständigen Positionsbestimmung der Knoten, des gemeinsamen Zeitverständnis und der eigenständigen Organisation weltweit untersucht. Die Anwendungsszenarien haben sich auch über den militärischen Bereich, wie z.B. die Gefechtsfeldüberwachung, hinaus erweitert. So können Einsatzmöglichkeiten dieser Technologie im Katastrophenschutz, der Gebäudeüberwachung, in der Logistik, der Biologie, der Medizin, im Sport und etlichen weiteren Bereichen gefunden werden. Um die Möglichkeiten und Vorteile von Sensornetzen jedoch effektiv nutzen zu können, muss dem Applikationsentwickler der Zugang zu dieser Technologie vereinfacht werden.

1.1 Motivation und Problemstellung

Im Vergleich zu Anwendungen für kabelgebundene Netze sieht sich der Entwickler einer Sensornetzapplikation neuen Rahmenbedingungen gegenüber. Kabelgebundene Netze bestehen meist zwischen Computern wie denen, die sich auf jedem Büroarbeitsplatz befinden. Diese Computer sind gleichzeitig auch an eine Energiequelle (gemeinhin eine Steckdose) gebunden und verfügen heutzutage über Speicherkapazitäten im mindestens zweistelligen Gigabytebereich. Kommunikation zwischen den Rechnern ist billig und auch Rechenleistung steht dank Zwei-, Vier- und Mehrkernprozessortechnologie schier unendlich zur Verfügung. Diese Rahmenbedingungen mit dieser Fülle an Ressourcen gelten jedoch nicht für Sensornetze.

Eine Hauptanforderung an Sensorknoten ist die möglichst geringe Größe. Die prototypischen Sensorknoten *Macro Motes*, die im Rahmen des oben genannten SMART DUST Projektes hergestellt wurden, sind ca. 100 mm^3. Aus diesen Dimensionsanforderungen

ergibt sich, dass Sensorknoten nur über geringe Speicher- und Rechenkapazität verfügen. Auch die Energieressourcen sind dadurch stark begrenzt. Insbesondere die Kommunikation über Funk benötigt jedoch viel Energie. Es wird schnell ersichtlich, dass bestehende algorithmische Lösungen für kabelgebundene Netze aufgrund der auf Sensorknoten knappen Ressourcen nicht auf Sensornetze übertragbar sind.

Zusätzlich zu den begrenzten Ressourcen der Sensorknoten ergeben sich neue Herausforderungen, die inhärent mit den Anwendungsszenarien der Sensornetze verknüpft sind. Sensorknoten sollen nach ihrer Ausbringung selbständig ein Netz aufbauen. Die Knoten verbinden sich eigenständig ad hoc mit ihren benachbarten Knoten. Im Vergleich hierzu gibt es bei kabelgebundenen Netzen eine Person (Administrator), welche die Netzstruktur (Netztopologie) festlegt und die einzelnen Computer dementsprechend konfiguriert.

Da Sensorknoten über Funk kommunizieren, sind sie nicht an einen festen Ort gebunden. In vielen Anwendungsszenarien (beispielsweise im Bereich der Logistik) ist die Mobilität der Knoten eine Grundvoraussetzung. Dies bedeutet jedoch, dass die Netztopologie ständiger Veränderung unterliegt, der sich die Knoten wiederum anpassen müssen.

Auch die Qualität der Funkübertragung unterliegt zeitlichen Schwankungen. Physikalische Einflüsse aus der Umgebung beeinflussen die Funkausbreitungscharakteristiken (wie beispielsweise die Reichweite) ständig. Ein Fahrzeug oder eine Person, die sich durch das Sensornetz bewegt, kann temporär dafür sorgen, dass eine Kommunikationsverbindung zwischen zwei Geräten nicht mehr vorhanden ist und somit eine veränderte Netztopologie vorliegt. Die genauen Charakteristiken der Funkübertragung sind ebenfalls Gegenstand aktueller Forschung im Bereich der Sensornetze.

Wie bereits erwähnt, sind Sensorknoten auf Grund ihrer Größe in ihren Ressourcen begrenzt. Dies bezieht sich insbesondere auf die Energieversorgung der Knoten. Zwar gibt es Ansätze, Sensorknoten mit autonomer Energieversorgung auszustatten, wie beispielsweise Solarzellen, jedoch besitzen die zu Zeit verfügbaren Knoten typischerweise einen Akkumulator mit begrenzter Kapazität als Energiequelle. Dies hat zur Folge, dass während des Betriebs eines Sensornetzes nach und nach einzelne Knoten ausfallen. Auch kann es passieren, dass einzelne Knoten durch äußere Einflüsse ausfallen. Dies kann beispielsweise in militärischen Szenarien oder im Katastrophenschutz vermehrt vorkommen, bei denen die Sensorknoten bisweilen extremer Hitze (Waldbrände), physikalischem Druck (schweres Geräte) oder Wasser (Überflutungen) ausgesetzt sind. Ausfälle einzelner Knoten können ebenfalls die Topologie eines Netzes drastisch beeinflussen.

Die Auswirkungen einer Änderung der Nachbarschaftsbeziehungen und somit einer Änderung der Topologie des Netzes werden deutlich, wenn man das Augenmerk auf die Hauptaufgabe eines Sensornetzes legt. Insbesondere sollen die einzelnen Knoten Daten über ihre Sensoren messen und diese an eine gemeinsame Datensenke durch das Netz weiterleiten. Die Kommunikation mit der Datensenke erfolgt jedoch nicht direkt. Aufgrund der nur begrenzt verfügbaren Energie verfügen Sensorknoten wie bereits erwähnt nur über eine begrenzte Funkreichweite, die sich nicht über das gesamte zu beobachtende Gebiet erstreckt. Aufgezeichnete Daten werden über andere Knoten entlang sogenannter *Pfa-*

de[1] zur Datensenke geleitet. Man spricht dabei von einer *Multi-Hop-Kommunikation*, wobei das Weiterleiten von einem Knoten zum direkten Nachbarn als *Hop* bezeichnet wird. Je länger der Pfad ist (je mehr Hops), desto größer ist auch die Wahrscheinlichkeit, dass es zu Fehlern aufgrund von nicht mehr existierenden Kommunikationsverbindungen zwischen zwei Knoten kommt. Je größer und häufiger diese Topologieveränderungen auftreten, desto schwieriger ist es, Daten über lange Pfade von einer Datenquelle zur Senke zu leiten. Eine effiziente (im Sinne des Kommunikationsaufwandes) und gleichzeitige robuste Wegewahl zu realisieren, ist daher eines der zu lösenden Probleme in Sensornetzen.

Da das Problem der Wegewahl innerhalb von Sensornetzen fundamental für die Nutzbarkeit dieser Technologie ist, gibt es bereits verschiedene vorgeschlagene Lösungen (*Algorithmen und Protokolle*), die in Kapitel 7.1 beschrieben werden. Ein *Algorithmus* ist eine bedingte Folge von Aktionen und ein *Protokoll* beschreibt dabei die Regeln zur Kommunikation zwischen Knoten, um das beschriebene Problem zu lösen.[2] Jedoch steht nach Meinung des Autors der Beweis der Praxistauglichkeit der meisten Protokolle noch aus. Die Gründe hierfür liegen, wie im Folgenden ausgeführt wird, zum einen in der vereinfachten Abstraktion des Problems auf graphentheoretische Probleme und zum anderen in dem Mangel an representativen Umsetzungen von Sensornetzen.

Netze lassen sich mathematisch als Graphen darstellen. Diese Form der Abstraktion wird oft bei der Entwicklung von Algorithmen und Protokollen zu Grunde gelegt. In einem Graphen repräsentieren die Knoten des Graphen die Sensorknoten des Netzes und die Kanten die existierenden Kommunikationsverbindungen zwischen den Knoten. So lassen die einzelnen Lösungsansätze sich an verschiedenen Netztopologien untersuchen. Bei der Übertragung in den Bereich der Graphentheorie gehen jedoch oben genannte Bedingungen insbesondere die zeitliche Veränderbarkeit der Topologie verloren. Erschwerend kommt hinzu, dass insbesondere die Funkausbreitungscharakteristiken nur schwer in eine mathematische Abstraktion übertragbar sind. Eine rein graphentheoretische Betrachtung kann daher nur der erste Schritt bei der Entwicklung eines Algorithmus oder eines Protokolls sein.

Um einen genaueren Einblick in die Funktionsweise von Algorithmen zu bekommen und deren Funktionsweise zu zeigen, werden Simulatoren benutzt, die Ad-hoc-Netze mit den einzelnen Knoten und den Funkverbindungen simulieren. Am weitesten verbreitet sind hier die Simulatoren *ns2* [19] und *GloMoSim* [129].[3] Am Institut für Telematik der Universität zu Lübeck wurde außerdem der Simulator *Shawn* [20] verwendet, der speziell zur Simulation von Netzen mit hoher Knotenzahl entwickelt wurde. Simulationen sind eine einfache Möglichkeit, schnell Ergebnisse zur Evaluation von Algorithmen zu bekommen. Die meisten Simulatoren sind frei verfügbar und erweiterbar. Eigene Algorithmen und Anwendungen können in einen Simulator integriert werden. Der Simulator kann diese Algorithmen in Szenarien verschiedener Größenordnungen evaluieren.

[1] Der Begriff *Pfad* wird im Kapitel 2 formal definiert.
[2] Die Begriffe *Algorithmus* und *Protokoll* werden in Kapitel 2 ebenfalls formal definiert.
[3] Die Daten zur Verbreitung der Simulatoren sind aus [50] entnommen.

Inwieweit die Resultate von Simulationen jedoch praxisrelevant sind, wird immer wieder in Frage gestellt. So wird in der Arbeit von Kotz, Newport und Elliot [48] festgestellt, dass die Annahmen, die den meisten Simulationen zu Grunde liegen, zu stark vereinfacht sind und die Diskrepanz zwischen Simulation und Realität so groß ist, dass viele durch Simulationen gewonnene Ergebnisse in Frage gestellt werden müssen. Hier werden insbesondere die folgenden Annahmen kritisiert: 1) Die Welt ist flach. 2) Die Funkausbreitung ist zirkulär (kreisrund). 3) Die Reichweite aller Funkmodule ist gleich. 4) Wenn ich dich höre, hörst du mich auch (Symmetrie). 5) Wenn ich dich höre, höre ich dich perfekt (Kein Verlust). 6) Die Signalstärke ist eine einfache Funktion abhängig von der Distanz. Es ist leicht ersichtlich, dass diese vereinfachten Annahmen Einfluss auf die Messergebnisse der Verfahren haben und die gewonnenen Erkenntnisse somit nur wenig Aufschluss über die Praxistauglichkeit der Algorithmen geben.

Neben den Problemen der Nachbildung der Realität in den Simulatoren weisen Kurkowski, Camp und Colagrosso [50] in ihrer Arbeit mit dem Titel *The Incredibles* von 2005 darauf hin, dass weniger als 15 % der auf der Konferenz MobiHoc [105] in dem Zeitraum zwischen 2000 und 2005 präsentierten Simulationsergebnisse reproduzierbar sind. Der Anteil der Veröffentlichungen, die Simulationen nutzen, belief sich dabei auf 75 % aller eingereichten Arbeiten. Die Tatsache, dass die Ergebnisse nicht reproduzierbar sind, ist laut den Autoren oft darauf zurückzuführen, dass die Simulationsparameter nicht vollständig angegeben seien. Zudem kann dadurch nicht sichergestellt werden, dass die in den Simulationen gewonnenen Ergebnisse statistisch korrekt sind. Insbesondere wird hier auf die Nutzung des Pseudozufallszahlengenerator von *ns2* hingewiesen, der für die Evaluation mehrerer Messdurchläufe unverzichtbar ist. Eine falsche Initialisierung kann hier zu statistischen Artefakten führen und somit die eigentliche Evaluation eines Algorithmus verfälschen. Auf diese Problematik wird nochmals in der Arbeit von Weigle [116] aus dem Jahre 2006 hingewiesen.

Implementierungen und Evaluationen auf echten (nicht simulierten) Sensorknoten sind selten. Dafür gibt es gleich mehrere Gründe: 1) finanzielle Gründe, 2) organisatorische Gründe und 3) zeitliche Gründe.

Zunächst muss die Hardware für Sensorknoten beschafft werden. Hier steht der Forscher vor der Möglichkeit, eigene Sensorknoten zu entwickeln oder auf bestehende Sensorknotenplattformen zurückzugreifen. In beiden Fällen sind die Kosten sehr hoch, da die Produktion von Sensorknoten nicht in Großserie betrieben wird. Um die Praxistauglichkeit von Algorithmen unter Beweis stellen zu können (gerade im Sinne der Skalierbarkeit), müssen die Evaluationsszenarien eher hundert als zehn Knoten umfassen.

An dieser Stelle gibt es neue organisatorische Herausforderungen, die bewältigt werden müssen. Die zur Evaluation benötigten Sensorknoten müssen über volle Energieressourcen (beispielsweise geladene Akkus) verfügen. Gleichzeitig muss jedes Gerät mit der zur Evaluation benötigten Software programmiert werden. Die Geräte müssen ausgebracht werden und während oder nach dem Experiment ausgelesen werden. Im Vergleich zu einer Simulation ist der organisatorische Aufwand für eine Messung mit Sensorknoten ungleich höher.

Neben dem organisatorischen Aufwand spielt auch die Zeit eine Rolle, die zur Implementierung von Algorithmen auf Sensorknoten benötigt wird. Da auf Sensorknoten nah an der Hardware programmiert wird, führen Programmierfehler meist zum Absturz des Programms, was wiederum zum Ausfall des Knoten führt. Weiterhin kann es zu sogenannten *Wettlaufsituationen* (engl.: *Race Conditions*) kommen, bei denen das Resultat der Aktionen von deren zeitlichen Auftreten abhängt. Diese Situationen ergeben sich zum einen durch die Nebenläufigkeit der parallelen *Ausführungsstränge* (engl.: *Threads*) auf einem Knoten und zum anderen durch die Parallelität in den Aktionen der verschiedenen Knoten. Diese Parallelität kann in Simulatoren häufig nicht nachgestellt werden, da hier Operationen in der Regel sequentiell abgearbeitet werden. Diese durch Wettlaufsituationen bedingten Fehler sind of nur schwer zu identifizieren und nur schwer zu beheben, da sie nicht immer reproduzierbar auftreten. Des Weiteren bietet ein Sensorknoten keine speziellen Möglichkeiten zum Auffinden von Programmierfehlern. Die Benutzerschnittstelle der verfügbaren Sensorknoten besteht meist aus zwei bis drei Knöpfen und einzelnen Leuchtdioden. In Ausnahmefällen verfügt ein Sensorknoten über eine serielle Schnittstelle. Der Status des Programms sowie die Werte einzelner Variablen sind jedoch dadurch nur schwer zu verfolgen. Die Rückmeldung für den frustrierten Programmierer ist in einem solchen Moment meist ein ausgeschaltetes Gerät, welches seine Gründe zur Dienstaufgabe nicht mehr preisgibt. Dies erschwert jedoch den Entwicklungsprozess von Algorithmen und Applikationen erheblich.

Legt man die Probleme der zu starken Abstraktion bei der graphentheoretischen Modellierung von Sensornetzen und der ungenügenden (wenn nicht gar unmöglichen) Nachbildung eines realen Sensornetzszenarios im Simulator sowie den Mangel an realen Umsetzungen von Sensornetzen zu Grunde, kommt man zu dem Schluss, dass viele grundlegende Probleme in Sensornetzen noch nicht praxistauglich gelöst sind.

Ob nun durch Bewegung, Funkübertragungseigenschaften oder durch Ausfall einzelner Knoten verursacht, so unterliegt ein Sensornetz im Gegensatz zu einem kabelgebundenen Netz potentiell ständigen Topologieänderungen. Gleichzeitig ist die Kommunikation der Knoten untereinander teuer (im Sinne der auf einem Knoten zur Verfügung stehenden Energie). Die Wegewahl ist dabei nur eine der Herausforderungen, die hier beispielhaft aufgeführt ist, der sich ein Entwickler von Sensornetzapplikationen gegenüber sieht. Die Sensornetztechnologie stellt jedoch wie bereits erwähnt ein Hilfsmittel für Forscher und Ingenieure in den verschiedensten Anwendungsgebieten dar. So möchte beispielsweise ein Biologe die Population einer Spezies in einem Biotop messen und verfolgen. Für ein solches Szenario würde sich die Nutzung eines Sensornetzes anbieten, doch benötigt der Biologie dafür wie oben beschrieben Expertenwissen über Sensornetze, deren Algorithmen und Hardware.

An dieser Stelle lassen sich Parallelen zu der Entwicklung des Computers ziehen. Anfang der 1960er Jahre, wurden die Rechner über Lochkarten oder Lochstreifen mit Programmen und Daten versorgt. Die Bedienung eines Rechners erforderte Computerfachkenntnisse seitens des Benutzers. Der Benutzer war gleichzeitig der Programmierer – diese Situation findet sich aktuell bei den Sensornetzen wieder. Auch hier stehen im Moment noch die Anwendungen im Vordergrund und nicht der Entwickler, dem es auf einfache

Weise ermöglicht werden muss, ohne speziellere Fachkenntnisse eine Applikation nach seinen Anforderungen zu erstellen.

Neben den bereits erwähnten netzbedingten Rahmenbedingungen, die ein Entwickler beachten muss, kommt erschwerend hinzu, dass eine Sensornetzapplikation inhärent eine *verteilte Applikation* ist, d.h. eine Applikation, die sich aus vielen Prozessen zusammensetzt, die auf verschiedenen Ressourcen (in diesem Fall die einzelnen Sensorknoten) laufen. Die einzelnen Prozesse koordinieren sich dabei lediglich über den Austausch von Nachrichten.

Die Problematik, die Erstellung von verteilten Anwendungen zu vereinfachen, wird bereits seit rund 20 Jahren untersucht. Die Grundidee ist, mittels einer sogenannten *Middleware* die Komplexität der Verteiltheit vor dem Anwendungsentwickler zu verbergen, d. h. transparent zu machen. Die Middleware kapselt beispielsweise die Kommunikationsvorgänge in einer separaten Schicht zwischen Betriebssystem und Anwendung. Verschiedene *Programmierparadigmen*[4]. prägen die Umsetzungen der Middleware. Während anfangs Sun RPC [98] entfernte Prozeduraufrufe nach dem prozeduralen Programmierparadigma realisierten, war es mit CORBA [78] und später mit Java RMI [97] möglich, verteilte Anwendungen objektorientiert zu implementieren. Mit steigender Komplexität wurde Mitte der neunziger Jahre der Begriff der *Service-orientierten Architektur* (SOA)[5] oder auch *dienstorientierten Architektur* als Paradigma zur Nutzung verteilter Funktionalität geprägt.[6] Die Grundidee der SOA ist dabei, jegliche Funktionalität als *Service* (Dienst) mit öffentlicher Schnittstelle anzusehen und Applikationen aus diesen Diensten zu komponieren (oder auch zu orchestrieren).

Anwendung finden Service-orientierte Architekturen insbesondere in Geschäftsanwendungen (engl.: *Enterprise Applications*). Hier findet sich eine große Verteiltheit, da einzelne Abteilungen verschiedene Aufgaben erfüllen. Zudem kann es passieren, dass neue Abteilungen eingegliedert werden oder andere ausgegliedert werden. Die Abstraktion der einzelnen Aufgaben als Dienste mit öffentlicher Schnittstelle und die Orchestrierung mittels einer Service-orientierten Architektur ermöglicht eine einfache Komposition von Software und erlaubt eine Flexibilität, die in der Unternehmenswelt notwendig ist, um wettbewerbsfähig zu bleiben.

Ein Sensornetz ist bedingt durch das zugrundeliegende Szenario ein *verteiltes System*, welches, wie bereits beschrieben, ständigen Änderungen durch physikalische Einflüsse unterliegt, an die es sich anpassen muss. Zudem können sich die Anforderungen des Benutzers an das Sensornetz mit der Zeit ändern. So interessiert den oben bereits erwähnten Biologen nun nicht mehr die Größe der Population einer Spezies, sondern das Bewegungsverhalten der einzelnen Individuen, welches durch das Sensornetz aufgezeichnet werden soll. Auch hier wird also eine hohe Flexibilität benötigt - gleichzeitig soll die Erstellung und Anpassung einer Sensornetzanwendung dem Anwender zugänglich gemacht werden.

[4] Auf die verschiedenen Programmierparadigmen wird in Kapitel 2 eingegangen.

[5] Der Begriff *Service-orientierte Architektur* ist ein feststehender Fachbegriff und wird daher im Folgenden groß geschrieben.

[6] Die Begrifflichkeiten zu SOA werden in Kapitel 2 formal eingeführt.

Neben der Erleichterung der Erstellung einer Sensornetzapplikation durch die Verwendung von Service-orientierten Architekturen soll auch der zunehmenden Bedeutung der Sensornetztechnologie im industriellen Bereich Genüge getragen werden. Hier ist ein Sensornetz als einzelner Baustein in einer Unternehmensanwendung zu sehen. Durch die Nutzung von Service-orientierten Architekturen können die einzelnen Funktionalitäten eines Sensornetzes als Geschäftsprozesse in einen darüberliegenden Geschäftsprozess eingebunden und somit flexibel in die im Unternehmen bestehende IT-Infrastruktur integriert werden.

Führt man den Gedanken der Service-Orientierung mit ihren Möglichkeiten zu Ende, eröffnet sich neben der vereinfachten Applikationserstellung und der Integration in bestehende Infrastrukturen ein dritter Aspekt, der sich auf die Sichtweise auf Sensornetzapplikationen bezieht. Wurde bisher meist eine Applikation für einen Sensorknoten geschrieben, welche dann auf alle Sensorknoten verteilt wurde, so kann man sich nun vorstellen, dass verschiedene Dienste nur auf bestimmten Knoten zur Verfügung stehen. Verschiedene Knoten übernehmen verschiedene Teilaufgaben, um somit die gestellte Gesamtaufgabe für das Sensornetz effizient zu erfüllen. Durch das einfache Hinzufügen oder Ersetzen von Diensten können dabei zusätzlich Strategien zur Selbstorganisation des Netzes erstellt werden. Basierend auf den Vorgaben des Entwicklers entscheiden die Sensorknoten selbständig, wer welche Dienste anbietet und somit wer welche Teilaufgaben erfüllt.

1.2 Zielsetzung

Wie bereits im vorangehenden Abschnitt erwähnt sieht sich ein Entwickler einer Sensornetzanwendung zwei Herausforderungen gegenüber: Zum einen benötigt er eine Infrastruktur, die es ihm erlaubt, auf abstrakter Ebene eine Anwendung für das Sensornetz zu formulieren, ohne dass er sich mit der Komplexität einer verteilten Sensornetzapplikation auseinandersetzen muss. Zum anderen benötigt er ein robustes Fundament von Algorithmen und Protokollen, die ihre Praxistauglichkeit unter Beweis gestellt haben und sich an sich ändernde Bedingungen im Sensornetz anpassen können.

Im Rahmen dieser Arbeit wird das service-orientierte Programmierparadigma auf Sensornetze übertragen und realisiert. Hierbei ist das Ziel, die Komplexität des Sensornetzes vor dem Anwendungsentwickler zu verbergen und diese Technologie somit auch Nicht-Sensornetzexperten zugänglich zu machen. Die Erstellung und Anpassung von Applikationen soll dabei derart vereinfacht werden, dass der Entwickler über keine besonderen Kenntnisse bezüglich Sensornetze verfügen muss. Zusätzlich soll dem Entwickler die Flexibilität durch die Austauschbarkeit einzelner Dienste gegeben werden, um auf einfache Weise seine Applikation an sich ändernde Anforderungen anzupassen. Sieht man beispielsweise einen Sensornetzforscher als Entwickler, trägt diese Flexibilität auch dazu bei, auf einfachere Weise verschiedene neue Algorithmen zu testen und zu evaluieren und somit auch den Fortschritt im Bereich der Sensornetzforschung zu beschleunigen. Die Umsetzung wird dabei so realisiert, dass die Möglichkeit der Selbstorganisation des Netzes besteht.

Zusätzlich werden verteilte Algorithmen vorgestellt, die nicht nur theoretisch und simulativ untersucht werden, sondern deren Praxistauglichkeit auch durch die Implementierung auf Sensorknoten unter Beweis gestellt wird. Die Algorithmen sind dabei unter den Aspekten der Robustheit und Adaptivität entwickelt worden. Hierbei wird neben einem Wegewahlalgorithmus ein Algorithmus vorgestellt, der die Selbstorganisation der Aufgabenverteilung basierend auf Vorgaben des Anwendungsentwicklers verwirklicht.

Diese Arbeit will in die Lücke zwischen den simulativ untersuchten Algorithmen und den Anforderungen, die eine konkrete Umsetzung stellt, vorstoßen. Der Beitrag wird darin gesehen, die Kluft zwischen den bisher in der Forschung theoretisch erworbenen Erkenntnissen und der realen Anwendung dieser Ergebnisse ein Stück zu schließen. Zu diesem Zweck wurden im Rahmen dieser Arbeit außerdem spezielle Sensorknoten entwickelt, die es erlauben, die neu entwickelte Infrastruktur und Algorithmen unter realistischen Bedingungen zu testen und zu evaluieren.

1.3 Gliederung der Arbeit

Im Anschluss an diese Einführung, werden in Kapitel 2 die Grundlagen zum weiteren Verständnis der Arbeit gelegt. Es werden sowohl fundamentale Begriffe zu Sensornetzen sowie zu Service-orientierten Architekturen erklärt und an technischen Umsetzungen beispielhaft beschrieben. Verwandte Arbeiten werden zu Beginn der jeweiligen Kapitel vorgestellt.

In Kapitel 3 werden die im Rahmen meines Forschungsprojekts *MarathonNet* entwickelten Sensorknoten *Pacemates* vorgestellt. Neben dem Hard- und Softwaredesign wird hier insbesondere auf die Anforderungen seitens der dem MarathonNet zugrundeliegenden Applikation sowie auf forschungsspezifische Anforderungen eingegangen.

In Kapitel 4 wird das Sensornetzbetriebssystem *Surfer OS* eingeführt. *Surfer OS* realisiert das service-orientierte Programmierparadigma auf den Sensorknoten, was die Basis zur Vereinfachung der Applikationentwicklung darstellt. Neben der dem System zu Grunde liegenden Idee und Philosophie, werden die einzelnen Komponenten von *Surfer OS* vorgestellt.

Kapitel 5 stellt ein im Rahmen dieser Arbeit entwickeltes Verfahren vor, mit welchem einzelne autonome Softwaremodule auf einen Sensorknoten migriert und in eine laufende Applikation eingebunden werden können. Das Verfahren wird auf der Sensorknotenplattform *Pacemate* realisiert und ermöglicht zudem auch statusbehaftete Migration der Module. Dem vorgestellten Verfahren liegt dabei der allgemeine Codeerstellungsprozess für C-Programme zugrunde, wodurch es sich ebenfalls auf andere Plattformen übertragen lässt.

In Kapitel 6 wird der *DySSCo*-Algorithmus vorgestellt, der Dienste basierend auf Vorgaben des Entwicklers selbsorganisierend auf die Knoten verteilt. Die Verteilung der Dienste passt sich dabei den Topologieänderungen des Netzes an, um die Funktionsweise

der Applikation zu gewährleisten. Mittels des *DySSCo*-Verfahrens wird dem Entwickler ermöglicht, auf abstrakte Weise die Anforderungen an seine Applikation zu beschreiben.

Kapitel 7 stellt das Wegewahlverfahren *GRAPE* vor, welches unter den Gesichtspunkten von Robustheit und Anpassbarkeit für drahtlose Netze im Rahmen dieser Arbeit entwickelt worden ist. Nach einer Vorstellung und Klassifizierung verschiedener existierender Wegewahlverfahren wird der algorithmische Ansatz und die Idee hinter *GRAPE* beschrieben. Das Verfahren wird sowohl simulativ wie auch auf den zuvor vorgestellten *Pacemate*-Sensorknoten evaluiert.

Kapitel 8 stellt die Infrastruktur vor, mittels derer sich basierend auf den Ergebnissen der vorangegangenen Kapitel Sensornetzanwendungen entwicklerorientiert, d.h. ohne Programmierkenntnisse und Wissen über verteilte Anwendungen und Sensornetze, entwerfen und umsetzen lassen. Dabei werden zunächst die Anforderungen an einen solchen entwicklerorientierten Designprozess und die damit verbundene Infrastruktur formuliert. Es wird die Architektur sowie die Implementierung der Infrastruktur beschrieben. An verschiedenen Fallbeispielen wird die Nutzung der Infrastruktur sowie der in den vorangegangenen Kapiteln erarbeiteten Konzepte demonstriert.

Die Arbeit schließt mit Kapitel 9, das die erzielten Ergebnisse zusammenfasst, einen Ausblick auf weitere Arbeiten gibt und Schlussfolgerungen für die zukünftige Entwicklung von Sensornetzapplikation formuliert.

2 Grundlagen

Nachdem in der Einleitung Sensornetze bereits motiviert und aus Anwendungssicht betrachtet worden sind, wird in diesem Kapitel die technisch formelle Sicht auf Sensornetze beschrieben. Insbesondere werden im Folgenden Begrifflichkeiten eingeführt, die Grundlage für spätere Definitionen sind. Weiterhin werden Applikationsszenarien für Sensornetze beschrieben, die verdeutlichen, wie vielfältig die Einsatzmöglichkeiten dieser Technologie sind. Zusätzlich werden einzelne Realisierungen von Sensornetzen exemplarisch genannt, die repräsentativ für den bisherigen Verlauf der Sensornetzentwicklung stehen.

Im zweiten Teil des Kapitels wird auf die Grundlagen des service-orientierten Programmierparadigmas eingegangen. Hier werden zunächst verschiedene Programmierparadigmen vorgestellt, gegenüber denen das service-orientierte Paradigma abgegrenzt wird. Die Begrifflichkeiten der Service-Orientierung werden eingeführt und das service-orientierte Paradigma wird motiviert. Abschließend wird beispielhaft die Technologie der Web Services zur Umsetzung einer Service-orientierten Architektur vorgestellt.

2.1 Sensornetze

Sensornetze stellen eine Technologie dar, mittels derer man Gebiete überwachen und Phänomene verfolgen kann. Die Vision ist, dass tausende autonome Kleinstcomputer in dem zu beobachtenden Gebiet ausgebracht werden und spontan ohne menschliche Administration über Funkkommunikation ein Netz bilden. Durch dieses Netz können die mittels der Sensorik der Sensorknoten aufgezeichneten Daten zu einer Datensenke geleitet werden, wo sie verarbeitet, interpretiert und aufbereitet werden. Dabei ergeben sich insbesondere durch die Verteiltheit des Systems und die Ressourcenbeschränkungen auf den Knoten neue Herausforderungen.

2.1.1 Begrifflichkeiten

In ihrer Anfangszeit waren Computersysteme zentralisiert. Es gab einen (Groß-)Rechner, der von mehreren Benutzern genutzt werden konnte. Es galt das Prinzip *Ein Computer, viele Benutzer*. Mit der Entwicklung des *Einzelplatzrechners* (engl: *Personal Computers, PC*) in den 70er Jahren galt nun das Prinzip *Ein Computer, ein Benutzer*. Im Moment geht die Entwicklung hin zu *Viele Computer, ein Benutzer*. Die einzelnen Computer kommunizieren untereinander und tauschen Nachrichten aus. Es entstehen sogenannte *Rechnernetze*. Tanenbaum [101] definiert den Begriff des Rechnernetzes als *[...] mehrere*

miteinander mit einer bestimmten Technologie verbundene autonome Computer [...]. Zwei Computer gelten als miteinander verbunden, wenn sie Informationen austauschen können.

Abstrakt betrachtet ist ein Rechnernetz ein sogenanntes *verteiltes System*, wo verschiedene autonome Rechner mittels Kommunikation miteinander interagieren. In der Literatur finden sich verschiedene Definitionen für ein verteiltes System, die sich dadurch unterscheiden, dass sie verschiedene Sichtweisen und Anforderungen an ein verteiltes System in den Vordergrund stellen. An dieser Stelle wird kurz auf die Definitionen eingegangen, da die verschiedenen Sichtweisen und Anforderungen ebenfalls an verschiedenen Stellen dieser Arbeit im Vordergrund stehen werden. Tanenbaum und Steen [102] legen in ihrer Definition Wert darauf, dass ein verteiltes System bestehend aus einer Ansammlung unabhängiger Computer dem Benutzer als ein einzelnes kohärentes System erscheint. Dies spiegelt sich auch im Ziel dieser Arbeit wieder, die Komplexität des Sensornetzes dem Programmierer zu verbergen. Coulouris et al. [15] definieren ein verteiltes System als ein System, in dem Hardware und Software-Komponenten auf vernetzten Rechnern nur mittels Austausch von Nachrichten kommunizieren können. In der theoretischen Betrachtung von verteilten Systemen betont Tel [103] in seiner Definition die Unabhängigkeit der in einem verteilten System interagierenden Computer, Prozesse oder Prozessoren. Eine Anwendung, die in einem verteilten System abläuft und dessen Eigenschaften nutzt, wird als *verteilte Anwendung* bezeichnet.

Basierend auf den oben genannten Defintionen für Rechnernetze und verteilte Systeme ist ein *Sensornetz* somit eine spezielle Ausprägung eines Rechnernetzes und stellt daher auch ein verteiltes System dar. Die Bestandteile eines Sensornetzes, die autonomen Computer, sind die *Sensorknoten* (im Folgenden auch *Knoten*). Sensorknoten sind im Vergleich zu anderen Computern möglichst klein. Abbildung 2.1 zeigt unterschiedliche Sensorknotenprototypen.

(a) Smart Dust (Quelle: [111])

(b) MICA2 (Quelle: [11])

(c) BTnode (Quelle: [6])

Abbildung 2.1: Verschiedene Sensorknoten

Die wenigen technischen Bestandteile eines Sensorknotens (die *Hardware*), wie Prozessor, Speicher, Kommunikationsschnittstelle und Sensor sind dabei nach Möglichkeit Standardbauteile, so dass diese Knoten möglichst günstig in großer Stückzahl produziert werden können. Sensornetze bestehen (je nach Anwendungsszenario) aus mehreren hun-

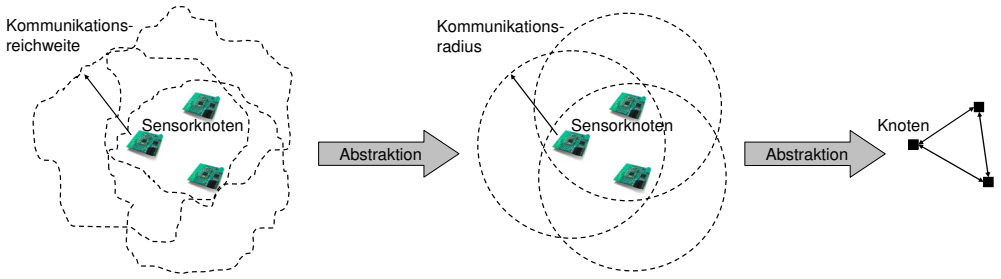


Abbildung 2.2: Ein gerichteter Graph als Abstraktion eines Netzes

derten bis tausenden dieser Knoten. Da die Sensorknoten speziell für eine Anwendung in Sensornetzen ausgelegt sind und Randbedingungen wie geringen Kosten und minimalem Platz folgen, gelten diese Geräte als sogenannte *eingebettete Systeme* (engl.: *embedded systems*).

Die Technologie, mittels derer die Sensorknoten Informationen austauschen bzw. sich verbinden, ist dabei die Funktechnologie. Der Austausch der Informationen über diese Kommunikationsverbindung oder diesen Kommunikationskanal kann dabei direkt oder über andere Knoten erfolgen. Das englische Synonym *Link* für Kommunikationskanal wird im Rahmen dieser Arbeit als direkte Kommunikationsmöglichkeit zwischen zwei Knoten ohne Zwischenknoten definiert.[1] Ein Link kann dabei *bidirektional* (ungerichtet) oder *unidirektional* (gerichtet) sein. Während im bidirektionalen Fall beide Knoten Informationen miteinander austauschen können, so kann im unidirektionalen Fall nur ein Knoten Informationen an den anderen Knoten senden. Diese Unterscheidung ist für Sensornetze von Bedeutung, da die Übertragungscharakteristiken des Funkes äußeren Störeinflüssen unterliegen, die sich örtlich verschieden auswirken können. Somit sind unidirektionale Links eine Charakteristik, die bei der Erstellung einer Sensornetzapplikation, Algorithmen und insbesondere Protokollen berücksichtigt werden muss. Ein Knoten kann einen anderen Knoten immer dann hören, wenn er in dessen *Kommunikationsreichweite* liegt. Bei vereinfachten Modellen wird angenommen, dass diese Kommunikationsreichweite keinen äußeren Einflüssen unterliegt und in alle Richtungen gleich ist. Daraus ergibt sich eine kreis- bzw. im dreidimensionalen Fall ein kugelförmige Funkausbreitung. Bei diesen vereinfachten Modellen wird daher auch von *Kommunikationsradius* gesprochen.

Mathematisch kann ein Netz als gerichteter Graph $G(V, E)$ dargestellt werden. Dabei beschreibt V die Menge aller Knoten (engl.: *vertices*) und E die Menge aller Kanten (engl.: *edges*). Eine Kante wird dabei als Paar (*2-Tupel*) von Knoten angegeben und beschreibt für das Netz, zwischen welchen Knoten ein Link existiert. Abbildung 2.2 zeigt einen gerichteten Graphen als Abstraktion eines Sensornetzes. Die Graphentheorie definiert einen Weg P von v_0 nach v_n durch einen Graphen G als n-Tupel von Knoten:

[1] In der Literatur wird der Begriff des Links meist nicht formal eingeführt. Gerade in Literatur über Internet und das World Wide Web ist ein Link implizit eine Kommunikation über mehrere Knoten.

$P = (v_0, ..., v_n)$. Dabei gilt: $\forall i \in [0...n-1] : (v_i, v_{i+1}) \in E$. Wege werden dabei unterschieden in Pfade, Zyklen und Kreise. Für Pfade gilt dabei zusätzlich zur Definition des Weges: $\forall i, j \in 0, ..., n \mid i \neq j : v_i \neq v_j$. Die Länge des Pfades ist dabei gegeben durch die Anzahl der Kanten und ist daher n. Da es für die Kommunikation keinen Sinn macht, eine Nachricht mehrfach über den selben Knoten zu leiten, wird von der Kommunikation innerhalb eines Netzes entlang von Pfaden gesprochen.[2] Ein Pfad gibt dabei an, über welche Knoten ausgehend vom ersten Knoten (der *Quelle*) die Information zum Zielknoten (der *Senke*) gelangt. Die Länge des Pfades wird auch als Anzahl von *Hops* bezeichnet und gibt dabei an, über wie viele Kanten die Informationen von der Quelle zur Senke geleitet werden.

Betrachtet man die Kommunikation aus der Sicht eines Knotens, so hat dieser Knoten Nachbarn, die über einen Link in einem Hop erreicht werden können und solche, die erst über mehrerer Hops erreicht werden. Als *n-hop-Nachbarschaft* eines Knotens bezeichnet man alle Knoten, die über maximal $n-1$ Zwischenknoten erreicht werden können. In Abbildung 2.3 sind verschiedene Nachbarschaften für einen Knoten dargestellt.

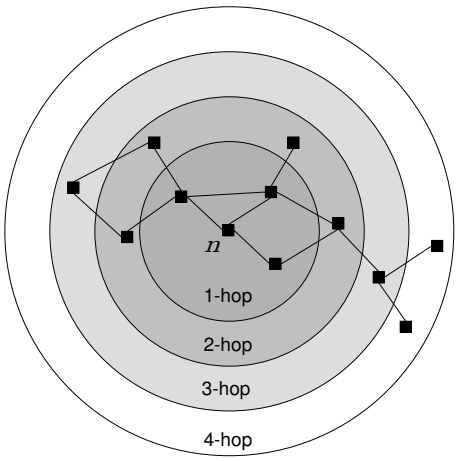


Abbildung 2.3: Verschiedene hop-Nachbarschaften des Knoten n in einem Netz

Wie bereits beschrieben stellt das Finden von Pfaden durch das Netz nur eines von vielen Problemen in Rechnernetzen dar. Verfahren zur Lösung von Problemen, die sich auf Computern implementieren lassen, werden *Algorithmen* genannt [94]. Man unterscheidet dabei zwischen *zentralisierten Algorithmen* und *verteilten Algorithmen*. Ein zentralisierter Algorithmus läuft in einem einzelnen Prozess ab und kann daher durch den sequentiellen Ablauf seiner Operationen beschrieben werden. Ein verteilter Algorithmus hingegen arbeitet auf einem verteilten System. Tel [103] definiert einen verteilten Algorithmus für eine Menge $\mathbb{P} = \{p_1, ..., p_n\}$ von Prozessen p_i als eine Menge zentralisierter Algorithmen mit je einem Algorithmus pro Prozess. Das Verhalten des verteilten Algorithmus wird

[2] Ob der mathematischen Definition von Pfaden bei der Kommunikation genüge getragen werden kann, hängt jedoch vom jeweiligen Wegewahlverfahren ab.

dabei beschrieben durch die Aktionen der Prozesse. Diese Aktionen ändern nicht nur den Status des Prozesses, sondern erzeugen auch Nachrichten und werden von Nachrichten beeinflusst werden. Im Unterschied zu einem zentralisierten Algorithmus gelten dabei für einen verteilten Algorithmus drei Aspekte: (1) kein globales Wissen, (2) keine globale Zeit und (3) Nichtdeterminismus [103]. Diese Aspekte ergeben sich aus den Charakteristiken eines verteilten Systems. Da die Kommunikation zwischen den Komponenten des verteilten Systems die Basis für die Zusammenarbeit darstellt, sind die verteilten Algorithmen den physikalischen Eigenschaften der Kommunikation ausgesetzt. Diese Eigenschaften sind beispielsweise Übertragungszeiten, Störungen sowie Kollisionen und können in der Regel nicht vorherbestimmt werden. Daher ist es auch unmöglich, die Sequenz der ausgeführten Operationen innerhalb des verteilten Algorithmus vorherzubestimmen, was den Entwurf von verteilten Algorithmen erschwert.

Um Nachrichten austauschen zu können, müssen Regeln für die Kommunikation festgelegt werden. Die Kommunikationspartner müssen sich über das Format, den Inhalt und die Bedeutung der gesendeten und empfangenen Nachrichten einig sein, damit ein erfolgreicher Austausch von Informationen stattfinden kann. Diese Regeln für die Kommunikation werden in sogenannten *Protokollen* formalisiert. Da der Austausch von Nachrichten Grundlage für einen verteilten Algorithmus ist (und meist sein wesentlicher Bestandteil), wird oft das Wort Protokoll als Synonym für einen verteilten Algorithmus verwendet. So wird von Wegewahlprotokollen (engl.: routing protocols), Zeitsynchronisationsprotokollen u. Ä. gesprochen.

Die Kommunikation in einem verteilten System wird gemäß dem durch die ISO (*International Organization for Standardization*) [39] standardisierten *OSI-Referenzmodell* (*Open Systems Interconnection*) [12] in sieben Schichten unterteilt. Dabei behandelt jede Schicht ein Teilproblem der Kommunikation zwischen zwei Geräten und bietet die Lösung der darüberliegenden Schicht im Form eines Dienstes an. So löst die unterste Schicht, die *Physikalische Schicht* unter anderem, wie die Übertragung einzelner Bits (0 und 1) zwischen zwei benachbarten Geräten abläuft und wie die Bits auf dem verwendeten Medium dargestellt werden. Die darüberliegende *Verbindungsschicht* regelt den Medienzugriff und überträgt die Bits unter Nutzung der Physikalischen Schicht. Mittels einer Prüfsumme wird hier sichergestellt, dass die einzelnen Bits korrekt übertragen werden. Erst auf der dritten Schicht, der *Netzwerkschicht*, werden die Wege durch das Netz mittels eines Wegewahlverfahrens bestimmt. Die darüberliegende *Transportschicht* bietet den über ihr liegenden Schichten Dienste zur sicheren Übertragung durch das Netz an. Auf dieser Schicht kann der Verlust von Paketen im Netzwerk registriert und gegebenenfalls darauf reagiert werden. Darüber befinden sich zudem die *Sitzungsschicht*, die *Präsentationsschicht* und die *Anwendungsschicht*. Die zuvor genannten Protokolle beschreiben dabei die Regeln, nach denen die entsprechenden Schichten zweier Gesprächspartner kommunizieren. Dies bedeutet, dass ein Protokoll und somit auch eine verteilter Algorithmus einer Schicht im ISO-OSI-Referenzmodell zugeordnet werden kann. Die individuellen Anforderugen einer Sensornetzapplikation betreffen dabei nicht nur die Anwendungsschicht. So ist das zu verwendende Wegewahlverfahren in der Netzwerkschicht abhängig von den Anwendungsparametern.

Abbildung 2.4: Abwurf von Sensorknoten über einem schwer zugänglichen Stadtgebiet (Quelle: angepasst aus [67])

2.1.2 Motivation und Applikationsszenarien

Wie bereits in der Einleitung beschrieben hat die Sensornetztechnologie ihre Ursprünge im militärischen Anwendungsgebiet. Die Möglichkeit ein Gebiet zu Überwachen, ohne dort präsent sein zu müssen, war zunächst insbesondere für das Militär von großem Interesse. So gibt es die Vision, dass ein Militärflugzeug Knoten über schwer zugänglichem Feindgebiet abwirft, wie es in Abbildung 2.4 dargestellt wird. Diese Knoten vernetzen sich selbständig untereinander und liefern Informationen über Truppenstärken und Truppenbewegungen. Die Hauptaufgaben, die ein Sensornetz erfüllen kann, gliedern sich dabei in die Überwachung (engl.: *monitoring*) und das Verfolgen (engl.: *tracking*) von Phänomenen. War in dem militärischen Szenario die Selbstorganization wichtig, um das Entsenden von Soldaten zur Administration des Systems zu vermeiden, so wurde jedoch schnell ersichtlich, dass diese Technik auch in anderen Gebieten nützlich ist. Die Tatsache, dass ein Sensornetz zur Überwachung und Verfolgung von Phänomenen und Ereignissen sich selbst organisiert und ohne weitere menschliche Interaktion agiert, eröffnete daher bald weitere Anwendungsszenarien. Die im Folgenden aufgeführten Beispiele sollen die Vielfältigkeit der Nutzung der Sensornetztechnologie verdeutlichen, stellen jedoch nicht den Anspruch, vollständig alle möglichen Szenarien abzudecken. Ein Überblick über verschiedene Anwendungsszenarien wird in Abbildung 2.5 gegeben. Neben den militärischen Anwendungen kann die Sensornetztechnologie ebenfalls in der Forschung, der Wirtschaft, der Industrie, der Medizin als auch im privaten Bereich von Nutzen sein und neue Möglichkeiten eröffnen [127].

So kann ein Forscher in der Biologie die Bewegungen und somit das Verhalten von Tieren beobachten, ohne selbst als Beobachter vor Ort zu sein und das Verhalten der Tiere zu beeinflussen und zu verfälschen. Hier kommt wieder die anfangs erwähnte Vision zu Tage, dass die Computer (hier die Sensorknoten) sich unsichtbar in die Umgebung einbinden.

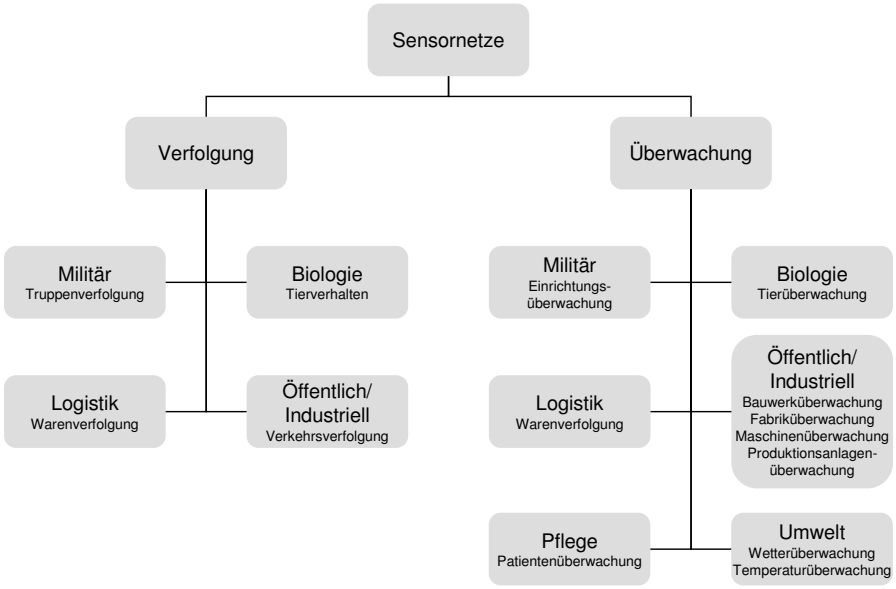


Abbildung 2.5: Verschiedene Anwendungsszenarien für Sensornetze

Als wirtschaftliche Anwendung lässt sich beispielsweise der Bereich der Logistik nennen. Einzelne Waren oder sogar Container an Verladeterminals könnten mit Sensorknoten ausgestattet werden. Jeder Container „wüsste“ somit über das Ziel seiner Reise Bescheid und könnte gegebenfalls eine Warnung senden, wenn die Container in seiner Nachbarschaft ein komplett anderes Ziel hätten. Dies könnte Abfertigungsprozesse beschleunigen und gleichzeitig Personal einsparen.

Eine Anwendung im industriellen Bereich wäre zum Beispiel die Überwachung von industriellen Anlagen. So kann in Produktionsstätten ohne zusätzlichen Personalaufwand mittels eines Sensornetzes genau beobachtet werden, ob Schäden auftreten oder etwaige Schadstoffe austreten. Die Sensorknoten können leicht ausgebracht werden, ohne zusätzliche Kommunikationsinfrastruktur zu installieren. Die erzeugten Daten können zudem automatisch ausgewertet werden und gegebenenfalls einen Alarm auslösen.

Als eine weitere Anwendung, die auch das private Leben vereinfachen kann, ist vorstellbar, dass die Bewegungen von Fahrzeugen verfolgt werden. Dadurch ließe sich der Verkehrsfluss immer aktuell analysieren, und es könnten störende Staus zu Stoßzeiten vermieden werden.

Im medizinischen Bereich könnte mit Hilfe eines Sensornetzes der Status von Patienten mittels Sensoren überwacht werden. Der Arzt könnte die Vitalparameter seiner Patienten jederzeit abrufen. Gleichzeitig könnte er informiert werden, wenn die Parameter einen gewissen Grenzwert überschreiten. Der Patient gewinnt dadurch wiederum eine gewisses Maß an Unabhängigkeit und kann sich trotz Überwachung frei bewegen.

Wie bereits erwähnt deckt diese Auflistung bei weitem nicht alle Anwendungsszenarien ab. Sie verdeutlicht aber wohl, dass auf den verschiedensten Gebieten Anwen-

Anwendung	Ziel	Knotenzahl	Dichte	Mobilität	Ausfall	Datenmenge
Truppenüberwachung	Verfolgung	groß	gering	statisch	hoch	gering
Tierverhalten	Verfolgung	groß	gering	statisch	gering	gering
Container	Überwachung	groß	hoch	mobil	gering	gering
Anlagenüberwachung	Überwachung	mittel	mittel	statisch	mittel	gering
Verkehrsüberwachung	Verfolgung	groß	gering	mobil	gering	hoch
Patientenüberwachung	Überwachung	mittel	hoch	mobil	gering	hoch

Tabelle 2.1: Beispielhafte Gegenüberstellung von Eigenschaften verschiedener Anwendungen

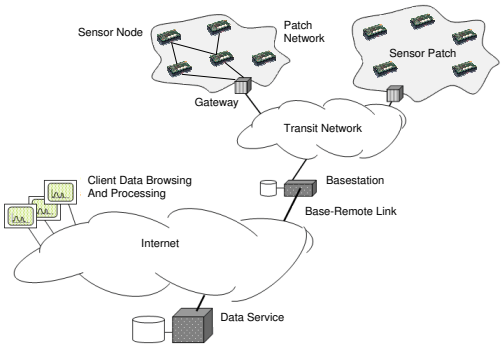
dungsmöglichkeiten zu finden sind. Außerdem wird deutlich, dass jede dieser Anwendungen ihre eigenen Anforderungen an das Sensornetz stellt. So muss in einigen Szenarien mit erhöhtem Ausfall von Geräten durch physikalische Einwirkungen gerechnet werden. In anderen Szenarien ist durch die Platzierung der Sensorknnoten damit zu rechnen, dass die zu erwartenden Funkeigenschaften nur schwer messbar oder vorhersagbar sind. Sind in einigen Szenarien die Knoten an Orten fixiert, so sind in anderen Szenarien große Änderungen in der Topologie durch Knotenbewegungen zu erwarten. Auch unterscheiden sich die Aufgaben der Sensornetze in den Szenarien. Werden in einigen Szenarien Daten aufgezeichnet und durch das Netzwerk zu einer Datensenke geleitet, so müssen in anderen beispielsweise einzelne Phänomene verknüpft und zugeordnet werden, um die Bewegung von Objekten durch das zu beobachtende Gebiet zu erkennen. Anhand dieser Erkenntnisse kann man die Anforderungen an Sensornetze in den verschiedene Anwendungsszenarien klassifizieren. Tabelle 2.1 gibt eine beispielhafte Übersicht über die Eigenschaften der verschiedenen oben genannten Anwendungen. Die Eigenschaften hängen dabei von dem Design der jeweiligen Anwendung ab und sind hier immer relativ zueinander zu sehen. An dieser Stelle sei nochmals auf eine Zielsetzung dieser Arbeit hingewiesen, nämlich es zu ermöglichen, flexibel auf diese Anforderungen zu reagieren.

2.1.3 Technische Umsetzung und Beispiele

Nachdem im obigen Abschnitt verschiedene Anwendungsszenarien vorgestellt worden sind, werden nun technische Umsetzungen aufgeführt. Dabei werden gleichzeitig die vielseitige Einsetzbarkeit von Sensornetzen, die Entwicklung der Sensornetztechnologie bis zum aktuellen Stand der Technik, sowie die Notwendigkeit für eine flexible Systemarchitektur, wie sie in dieser Arbeit präsentiert wird, verdeutlicht. Im Folgenden werden die Projekte *Great Duck Island*, *CodeBlue* und *PAWS* vorgestellt.

Einer der ersten Einsätze eines Sensornetzes, über den publiziert wurde, kommt aus dem Bereich der Biologie. Die Ergebnisse wurden auf dem *1. ACM Workshop on Wireless Sensor Networks and Applications* im Jahre 2002 [71] veröffentlicht. In einer Zusammenarbeit zwischen dem Intel Research Laboratory und der University of California in Berkeley wurde das Brutverhalten von Seevögeln auf der *Great Duck Island* im US-Bundesstaat Maine mit Hilfe eines Sensornetzes untersucht (siehe Abbildung 2.6(a)). Die Sensorknoten sollten dabei Feuchtigkeit und Wärme messen. Im ersten Einsatz wurden 32 Sensorknoten ausgebracht, von denen neun in den unterirdischen Brutstätten der Seevögel untergebracht wurden. Die übrigen wurden überirdisch verteilt. Die Kommunikation dieser statischen Knoten verlief direkt zu einen speziellen sogenannten *Gateway*-Knoten, der mit einem eingebetteten Linux Betriebssystem betrieben wurde. Dieser Gateway-Knoten übermittelte die Daten an eine Basisstation, die die aufgezeichneten Daten über das Internet in eine Datenbank übertrug. Die Architektur ist in Abbildung 2.6(b) dargestellt.

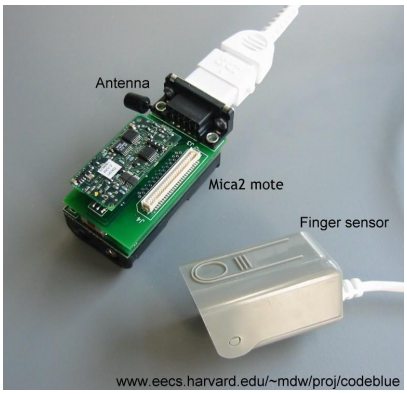
(a) Student untersucht Seevogel auf Great Duck Island (Quelle: http://berkeley.edu/news/media/releases/2002/08/05_snsor.html)



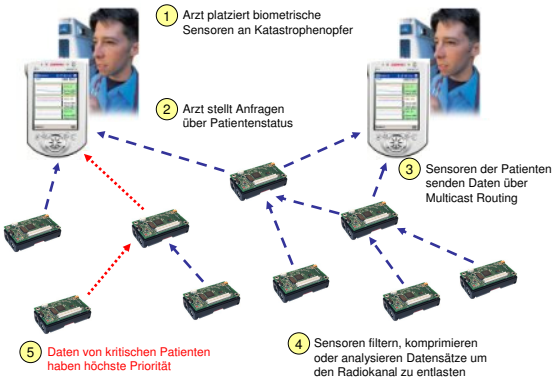
(b) Architektur des Sensornetzes auf Great Duck Island (Quelle: [71])

Abbildung 2.6: Einsatz eines Sensornetzes zur Überwachung des Brutverhaltens von Seevögeln auf Great Duck Island

In einem zweiten Einsatz auf Great Duck Island [99] im Jahr 2003 wurden zwei Sensornetze ausgebracht. Dabei wurde das eine Sensornetz mit 49 Sensorknoten nur im *one-hop-Modus*, also mit direkter Kommunikation zum Gateway, betrieben und das zweite Sensornetz mit 98 Sensorknoten im *multi-hop-Modus* betrieben. Während des viermonatigen Einsatzes erzeugte das Sensornetz über 650.000 Datensätze. Die Lebensdauer der Knoten war jedoch insbesondere im multi-hop-Modus im Schnitt unter 45 % der zuvor berechneten Lebensdauer von 80 bis 90 Tagen. Fast 25 % der Knoten im multi-hop-Modus hatten eine Lebensdauer von weniger als 25 Tagen. Des Weiteren wurden im multi-hop-Modus im Mittel nur 58 % der Nachrichten ausgeliefert, bei einer maximalen Pfadlänge von 7 Hops. Aus diesen Ergebnissen gewann man die Erkenntnis, dass eine ein-



(a) Sensorknoten mit Pulsoximeter (Quelle: [68])



(b) Architektur des CodeBlue-Sensornetzes (Quelle: übersetzt von http://fiji.eecs.harvard.edu/CodeBlue)

Abbildung 2.7: Einsatz des Sensornetzes CodeBlue zur Überwachung von Patienten

fache *sense-and-send*-Anwendungen, also ein sofortiges Senden nach dem Auslesen der Sensordaten, für ein Multi-hop-Sensornetzwerk nicht geeignet ist. Desweiteren erkannte man, dass oft Nachrichten über Knoten gesendet wurden, deren Batteriekapazitäten bereits geringer waren als bei anderen Knoten im Netz. Die letzte Aussage, die in dieser Veröffentlichung getroffen wird, besagt, dass ein Großteil des Verhaltens eines Sensornetzes nicht beobachtet werden kann, ohne ein komplettes System zu erstellen und es am Einsatzort zu installieren. Daraus lässt sich schließen, dass viele Netzwerkcharakteristiken, die notwendig für einen erfolgreichen Betrieb eines Sensornetzes sind, sich nicht im Vorfeld durch Berechnungen, Modellbildungen und Simulationen bestimmen lassen.

Im Jahr 2004 stellte eine Arbeitsgruppe der Universität Harvard das Projekt *CodeBlue* [72][68] vor. Bei CodeBlue handelt es sich um ein Sensornetz für medizinische Anwendungen. Dabei soll CodeBlue eine schnelle Reaktion in medizinischen Notfällen sowohl innerhalb eines Krankenhauses aber auch außerhalb, d. h. während der Pflege, ermöglichen. Zudem soll insbesondere die Pflege von Herzpatienten unterstützt werden. Die verwendeten Sensoren zeichnen dabei Vitaldaten, wie Herzfrequenz, Blutdruck, Sauerstoffsättigung des Blutes und ein Elektrokardiogramm des Patienten auf. Abbildung 2.7(a) zeigt einen Sensorknoten mit Pulsoximeter, wie er in CodeBlue verwendet wird. Geraten die Vitalparameter des mit Sensoren ausgestatteten Patienten in zuvor definierte Grenzbereiche, wird der zuständige Arzt alarmiert. Durch ein zusätzliches funkbasiertes Lokalisationsverfahren kann dem Arzt zudem die aktuelle Position des Patienten übermittelt werden. Dadurch kann CodeBlue eine ständige Überwachung vieler Patienten mit wenig Personal bei gleichzeitiger schneller und gezielter Hilfe ermöglichen. Die CodeBlue-Architektur, dargestellt in Abbildung 2.7(b), muss sich dabei zwei besonderen Herausforderungen stellen. Im Vergleich zu dem Sensornetz auf Great Duck Island handelt es sich bei CodeBlue um ein mobiles Sensornetz. Die Patienten können sich frei bewegen, was zu ständigen Änderungen in der Topologie führt. Gleichzeitig darf es aber zu keinem Verlust von Daten kommen. Die zweite Herausforderung ist die Lo-

kalisation der Patienten. In einem technischen Report der CodeBlue-Arbeitsgruppe aus Harvard [95] werden Ergebnisse eines Einsatzes von 30 Sensorknoten über drei Etagen eines Gebäudes beschrieben. Dabei wird aufgezeigt, dass ein einfaches Senden von Daten bei einer durchschnittlichen Pfadlänge von 5 Hops zu einem Verlust von knapp 40 % der Daten führt. Ein mehrmaliges Senden der Daten würde diesem Verlust für niedrige Datenraten entgegenwirken, jedoch bei höheren Datenraten zu größeren Verlusten führen, die auf die begrenzten Kapazitäten des Funkmediums zurückzuführen sind. Als ideale Lösung wird in dieser Arbeit vorgeschlagen, dass die Sensorknoten ihr Sendeverhalten an die lokalen Gegebenheiten wie Störeinflüsse auf dem Funkmedium anpassen müssten.

Das *PAWS*-Projekt der Universität Cape Breton, Kanada [7] hat zum Ziel, Öl- und Gasförderungsplattformen mittels drahtloser Sensornetze zu überwachen. PAWS steht dabei für *Petroleum Application of Wireless Systems*. Die Verkabelung industrieller Großanlagen kann Kosten zwischen 200 US-Dollar und 2.000 US-Dollar pro Meter verursachen. Der Einsatz drahtloser Sensornetze kann dabei die Kosten drastisch verringern. Jedoch stellt hier der Umgang mit den Umgebungsbedingungen eine noch größere Herausforderung dar. Schwankende Feuchtigkeitspegel und Vibrationen sorgen für erhöhten Ausfall einzelner Knoten. Zudem erschwert vor allem die Menge an Stahlkonstruktionen die Kommunikation zwischen den Knoten, da diese die verschiedensten Auswirkungen auf die Funkübertragungseigenschaften haben können. In einer Veröffentlichung von 2007 [44] werden dabei erste Erfahrungen einer Testevaluation mit 4 Sensorknoten präsentiert. Dabei wurden die empfangenen Funksignalstärken (engl.: *Received Signal Strength Indication* (RSSI)) und das Rauschen auf dem Funkmedium gemessen. Die Ergebnisse zeigen, dass die Störungen auf dem Medium hervorgerufen durch die Charakteristiken einer Industrieanlage die Leistungsfähigkeit des Sensornetzes stark beeinflussen. In der Arbeit wird ebenfalls darauf hingewiesen, dass diese Charakteristiken zunächst vor Ort gemessen werden müssen, bevor eine Anwendung entworfen und erfolgreich eingesetzt werden kann.

Die aufgeführten Beispiele belegen, dass der Bedarf für die drahtlose Sensornetztechnologie in verschiedenen Anwendungsgebieten vorhanden ist. Zudem zeigen die Beispiele aber auch deutlich, dass besonderes Expertenwissen erforderlich ist, bevor ein Sensornetz zum Einsatz gebracht werden kann. So müssen Umgebungsbedingungen zuvor am Ort des Einsatzes vermessen werden, Datenraten müssen abgeschätzt werden, Topologieänderungen hervorgerufen durch Ausfall oder bestimmte Bewegungsmuster der Knoten müssen modelliert werden und vieles mehr. Selbst nach diesen aufwendigen Arbeiten, die von hochqualifiziertem Personal durchgeführt werden müssen, kann nicht garantiert werden, ob das Sensornetz die gestellte Aufgabe erfolgreich ausführt. Außerdem bedingen Anpasssungen seitens der Benutzeranforderungen unter Umständen einen erneuten Entwurf der Anwendung. Hier ist ein Paradigma erforderlich, das eine hohe Flexibilität des Systems auch während des Einsatzes ermöglicht, um auf die vor Ort herrschenden Bedingungen reagieren zu können. Weiterhin möchte der Anwender die Möglichkeit haben, seine Applikation flexibel anpassen zu können.

2.2 Service-orientierte Architekturen

Mit der Fertigstellung der ersten Digitalrecher Anfang der 1940er Jahre entwickelte sich bald die Forschung an Sprachen, mittels derer man Abfolgen von Berechnungsschritten beschreiben kann. Die *Zuse Z3* von Konrad Zuse aus dem Jahre 1941 sowie der *Mark I* aus dem Jahre 1943 entwickelt von Howard H. Aiken von der Universität Harvard wurden mit Lochstreifen programmiert. Konrad Zuse entwickelte jedoch bereits 1945 die Sprache *Plankalkül*, die es erlaubte, Folgen von Berechnungen und ganze Algorithmen zu beschreiben. Die Sprache wurde jedoch wegen der Ereignisse des Zweiten Weltkrieges nicht implementiert und erstmalig 1972 veröffentlicht. Bis Mitte der 1950er Jahre wurden daher Programme direkt im Maschinencode des jeweiligen Computers geschrieben. Dies bedeutet, dass alle Instruktionen als numerische Codes eingegeben wurden. Wegen absoluter Adressierung des Arbeitsspeichers innerhalb der Programme war es außerdem sehr aufwendig, zusätzliche Instruktionen nachträglich in ein Programm einzufügen. Die Programme waren daher nur schwer lesbar und erforderten genaueste Kenntnis über die Hardware. Mitte der 1950er Jahre wurde von IBM mit *FORTRAN* (The IBM Mathematical *FOR*mula *TRAN*slating System) die erste höhere Programmiersprache für den IBM 704 Computer implementiert. FORTRAN wurde als Programmiersprache für wissenschaftliche Nutzung entwickelt. Zudem wurde sie in einer Zeit entwickelt, in welcher der Computer wesentlich teurer war als der Programmierer und somit der Fokus auf einer effiziente Nutzung der Rechnerressourcen lag und weniger auf einer einfachen Erlernbarkeit. Mit *COBOL* aus dem Jahre 1960 wurde erstmals eine Sprache entwickelt, die für eine Zielgruppe von Anwendern außerhalb der Informatik enstanden ist. COBOL steht dabei für *CO*mmon *B*usiness *O*riented *L*anguage, war speziell für kaufmännische Anwendungen im betriebswirtschaftlichen Bereich entwickelt worden und wird dort auch heute noch verwendet. Die Syntax von COBOL orientiert sich dabei an der natürlichen Sprache, um leichter lesbaren und verständlichen Programmcode zu erhalten. Mit *Basic* (*B*eginner's *A*ll-purpose *S*ymbolic *I*nstruction *C*ode) wurde 1964 am Dartsmouth College eine weitere Sprache veröffentlicht, die zum Ziel hatte, für Nichtinformatiker leicht erlernbar zu sein. Ursprünglich war Basic für die Studenten der Freien Künste des Colleges entwickelt worden, doch insbesondere durch den Aufstieg der Heimcomputer Ende der 1970er Jahre wurde die Sprache weit verbreitet.

Jeder Programmiersprache liegt ein sogenanntes *Programmierparadigma* zugrunde. Ein Programmierparadigma beschreibt nach welchem Prinzip mittels der Programmiersprache oder Programmiertechnik eine Aufgabe gelöst wird. Die Sprache Basic folgt dabei in ihrer ursprünglichen Form dem Paradigma der *imperativen* Programmierung. Dabei wird ein Programm als Abfolge von Befehlen gesehen, die der Rechner in einer definierten Reihenfolge abarbeitet, um eine Aufgabe zu lösen. Den übrigen bis hierher aufgeführten Programmiersprachen (abgesehen von dem Maschinencode) liegt das *prozedurale* Programmierparadigma zugrunde. Bei diesem Paradigma wird die zu lösende Aufgabe in kleinere Teilaufgaben (sogenannte Prozeduren) zerlegt. Dies soll zu einer einfacheren Strukturierung der Programme führen und Teile des Programmcodes wiederverwendbar machen.

Mit *Smalltalk* wurde in den 1970er Jahren eine Sprache entwickelt, die dem *objektorientierten* Programmierparadigma folgt. Mittels der Objektorientierung sollte eine Abbildung der realen Welt, die aus der Interaktion von realen Objekten besteht, in ein Programm vereinfacht werden. Einzelne Objekte kapseln ihre Daten und stellen ihre Funktionalität über Methoden anderen Objekten zur Verfügung. Die Programme können so durch eine Anpassung der Objekte leicht geändert werden. Das Paradigma der Objektorientierung liegt ebenfalls den später entwickelten Sprachen *C++* und *Java* zugrunde.

Im Zuge der Entwicklung des Internets und des World Wide Webs (WWW) gewannen verteilte Anwendungen immer größere Bedeutung. Insbesondere im Geschäftsbereich konnten nun verschiedene Unternehmensbereiche über das Internet direkt miteinander verknüpft werden. Auch geschäftsübergreifende Anwendungen (engl.: *Business-to-Business*) sind dabei von großem Interesse. Um diese immer komplexer werdenden Interaktionen strukturieren und flexibel gestalten zu können, entwickelte sich Mitte der 1990er Jahre das Paradigma der *Dienstorientierung* oder auch *Service-Orientierung*, das im Folgenden genauer beschrieben wird.

2.2.1 Begrifflichkeiten

Im alltäglichen Leben stellt jede Person, die mit ihren Fähigkeiten eine Aufgabe zur Unterstützung von anderen ausführt, praktisch einen Dienst zur Verfügung. Eine Gruppe von Personen stellt durch Ausführung von größeren Aufgaben wiederum einen Dienst für andere zur Verfügung. Dieses Prinzip zur Lösung von Aufgaben aus dem alltäglichen Leben wird mittels des Paradigmas der *Dienstorientierung* (*Service-Orientierung*) in die Informatik übertragen.

Die Basiseinheit der Service-Orientierung bildet dabei der *Dienst* oder auch *Service*. In dieser Arbeit werden die Begriffe *Dienst* und *Service* sowie *Dienstorientierung* und *Service-Orientierung* synonym verwendet. Ein Service liegt dabei als unabhängiges Softwareprogramm vor, welches lokal oder über ein Netzwerk von anderen genutzt werden kann. Die *OASIS* (Organization for the Advancement of Structured Information Standards), ein Konsortium aus über 600 Firmen zur Standardisierung von Basistechnologien für Informationssysteme, definiert in ihrem Referenzmodell für Service-orientierte Architekturen [80] einen Service als einen Mechanismus, der Zugang zu Fähigkeiten bzw. Funktionalitäten ermöglicht. Die Funktionalitäten eines Services ruft man dabei über *Serviceschnittstellen*, die in einer sogenannten *Servicebeschreibung* (engl.: *Service Description*) beschrieben werden. Diese Servicebeschreibung muss dabei in einer für andere Services verständlichen Form vorliegen und sollte aus Gründen der Interoperabilität unabhängig von der verwendeten Programmiersprache oder der ausführenden Plattform sein. Die auf einem Gerät ausgeführte Implementierung eines Services wird in dieser Arbeit mit dem Begriff *Serviceinstanz* bzw. *Dienstinstanz* beschrieben.

Bei der Interaktion mit einem Dienst gibt es drei Rollen: den Anbieter (engl.: *Service Provider*), den Nutzer (engl.: *Service Consumer*) und den Vermittler bzw. das Dienstverzeichnis (engl.: *Service Directory*). Das Zusammenspiel zwischen den drei Rollen ist in

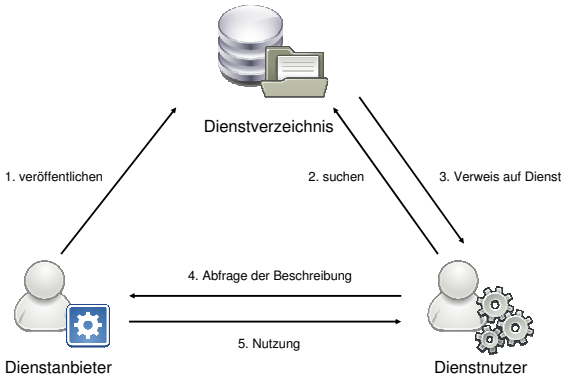


Abbildung 2.8: Zusammenspiel zwischen den einzelnen Rollen bei der Service-Orientierung

Abbildung 2.8 dargestellt. Der Anbieter eines Dienstes veröffentlicht dabei den von ihm angebotenen Dienst über das Dienstverzeichnis. Der Dienstnutzer sucht in dem Dienstverzeichnis nach einem bestimmten Dienst und erhält einen Verweis auf den Dienst und dessen Dienstbeschreibung. Der Nutzer erfragt Dienstbeschreibung unter der erhaltenen Adresse und nutzt anschließend die Funkionalitäten dieses Dienstes.

Dem Nutzer eines Dienstes bleibt durch die Service-Beschreibung die Implementierung des Dienstes seitens des Anbieters verborgen. Der Nutzer erhält lediglich die Informationen aus der Service-Beschreibung, wie der Service aufgerufen wird. Dies beinhaltet insbesondere, welches Protokoll und Nachrichtenformat verwendet werden muss, um den Service korrekt aufzurufen. Nutzt ein Service einen anderen, so spricht man von einer *Komposition* von Services bzw. einer *Servicekomposition* oder auch *Orchestrierung*. Verfolgt man das Paradigma der Service-Orientierung konsequent, so wird eine gesamte Applikation durch die Komposition verschiedener Services erstellt.

In dieser Arbeit beschreibt eine service-orientierte Infrastruktur zusammenwirkende Komponenten, welche die Nutzung von Diensten im Sinne der Service-Orientierung ermöglichen oder unterstützen.

Beschreibt das service-orientierte Programmierparadigma als Lösungslogik eine Theorie aus abstrakten Elementen wie Services, Service Provider und Service Consumer, so bezeichnet man eine technologische Umsetzung des service-orientierten Paradigmas als *Service-orientierte Architektur* kurz SOA. Eine SOA ist somit eine Technologiearchitektur, die das Erstellen von Anwendungen mittels der service-orientierten Lösungslogik ermöglicht. In der Literatur finden sich verschiedene Definitionen einer SOA, die sich im Wesentlichen im Grad ihrer Generalität unterscheiden. Die Definition der OASIS aus dem Englischen übersetzt lautet dabei: *[...] Eine SOA ist ein Netz aus unabhängigen Diensten, Maschinen, Menschen, die diese Dienste betreiben, beeinflussen, nutzen und steuern, sowie aus den Lieferanten von Ausrüstung und Personal für diese Menschen und Dienste [...]* [79]. Da diese Definition sehr allgemein und wenig technisch orientiert ist, sei an dieser Stelle eine zweite speziellere Definition von Melzer et al. [2] gegeben:

Unter einer SOA versteht man eine Systemarchitektur, die vielfältige, verschiedene und eventuell inkompatible Methoden oder Applikationen als wiederverwendbare und offen zugreifbare Dienste repräsentiert und dadurch eine plattform- und sprachenunabhängige Nutzung und Wiederverwendung ermöglicht. Beschreibt die Definition der OASIS die strukturellen Eigenschaften, nämlich Bestandteile und Komponenten einer SOA, so gehen Melzer et al. auf die eigentlichen Ziele einer SOA ein, die im Folgenden im Detail beschrieben werden.

2.2.2 Motivation

Die Entwicklung des service-orientierten Programmierparadigmas wurde insbesondere durch das Feld der Geschäftsanwendungen vorangetrieben. Inbesondere in den letzten Jahren werden dabei immer größer Anforderungen an die Unternehmens-IT gestellt bedingt durch die zunehmende Dynamisierung und Globalisierung der Geschäftswelt. Bisher waren Geschäftsanwendungen in sich abgeschlossene Komplettlösungen maßgeschneidert für die Anfordungen eines Unternehmens. Mittels genauer Kenntnis der Unternehmensstrukturen und standardisierter Entwurfsverfahren können schnell effiziente und genau angepasste IT-Lösungen erstellt werden. Jedoch unterliegt die Geschäftswelt einem ständigen Wandel. Umstrukturierungen, Zusammenlegungen und Outsourcing können die Abläufe in einem Unternehmen komplett verändern. Diese Flexibilität, die ein Unternehmen besitzen muss, sollte daher auch seitens der IT-Infrastruktur gegeben sein. Mit der immer größer werdenden Bedeutung von IT in Unternehmen und der dadurch steigenden Komplexität sind Anpassungen bei solchen in sich abgeschlossenen Anwendungen mit hohem Aufwand verbunden. Auch die Integration und Erweiterung einer Geschäftsanwendung beim Wachstum eines Unternehmens ist dadurch zunehmend schwieriger. Insbesondere bei der Zusammenlegung stellt die Integration der verschiedenen innerhalb der Geschäftsteile verwendeten Technologien ein Problem dar. Es wird zusätzliche Programmlogik benötigt, um beispielsweise Daten zwischen den jeweiligen Softwarekomponenten austauschen zu können.

Durch die Globalisierung und unterstützt durch die Entwicklung des WWW ergibt sich eine zweite Anforderung an die Unternehmens-IT. Man möchte IT-Strukturen geschäftsübergeifend verbinden. So können Informationen weltweit verknüpft werden und in die Geschäftsprozesse eingebunden werden. Auch hier sieht man sich dem Problem verschiedener verwendeter Techniken gegenüber. Zudem kann man mit zunehmender Dynamisierung der Geschäftswelt nicht vorhersehen, mit welchem Geschäftspartner man in nächster Zeit in Kontakt treten wird.

Um der Dynamik und Vernetzung der Geschäftswelt, die sich somit auch in der IT der Unternehmen widerspiegelt, Herr zu werden, haben sich die Anforderungen an die verwandten Techniken verändert. Im Fokus stehen nun die Aspekte der vereinfachten Anpassbarkeit und der verbesserten Interoperabilität, die durch bisherige Programmierparadigmen nicht hinreichend erfüllt wurden. Das Paradigma der Service-Orientierung versucht diese Anforderungen zu erfüllen, indem verschiedene Prinzipien verfolgt werden.

Mittels einer SOA wird durch die Aufteilung in einzelne Dienste eine Theorie aus der Softwareerstellung umgesetzt, welche als *Trennung von Anforderungen* (engl.: *Separation of Concerns*) bezeichnet wird. Diese Theorie besagt, dass ein großes Problem leichter gelöst werden kann, wenn es in eine Reihe kleinerer Probleme oder Anforderungen zerlegt wird. Dadurch möchte man die Komplexität der modernen verteilten Anwendungen beherrschen.

In einer SOA werden Dienste *lose gekoppelt* (engl.: *loose coupling*). Unter einer losen Kopplung versteht man, dass nur geringe Abhängigkeiten zwischen den Einheiten der Lösungslogik (in diesem Fall zwischen den Diensten) besteht. In einer SOA heißt dies, dass Dienste von der Anwendung oder anderen Diensten erst bei Bedarf dynamisch gesucht und eingebunden werden. Dies bedeutet, dass zum Zeitpunkt der Übersetzung eines Programms noch nicht bekannt ist, welche Dienstinstanz aufgerufen wird. Somit erreicht man eine hohe Flexibilität innerhalb des Programms. Es können leicht Dienste ersetzt oder geändert werden. Auch die Abfolge von Dienstaufrufen kann leicht an sich ändernde Arbeitsabläufe angepasst werden. Durch die Abstraktion von Funktionalitäten als Dienste erreicht man, das vorhandene Lösungslogik einfacher wiederverwendet wir.

Dadurch, dass Dienste eigenständige Programme mit definierten Schnittstellen darstellen, erreicht man eine Kapselung von Informationen. Die zugrundeliegenden Entwurfs- und Implementierungsdetails werden verborgen. Insbesondere spielt es dadurch keine Rolle welche Programmiersprache, welches Betriebssystem oder welche Hard- und Software für den Dienst verwendet wird. Dies erhöht die Übersichtlichkeit und ermöglicht erst die Komposition verschiedener Dienste. Durch die Verwendung von offenen Standards erreicht man zusätzlich Interoperabilität zwischen verschiedenen Plattformen.

Bisher wurde in diesem Abschnitt auf Service-Orientierung und SOA in Verbindung mit Unternehmensanwendungen als treibende Kraft eingegangen. Wie aber in Kapitel 1.1 und im vorangehenden Abschnitt verdeutlicht wurde, gelten die Aspekte der Flexibilität, Anpassbarkeit und Interoperabilität inbesondere auch im Bereich der Sensornetze. Hier wird mit dieser Arbeit ein neues Gebiet betreten, indem das Paradigma der Service-Orientierung auf Sensornetze übertragen wird.

2.2.3 Technische Umsetzungen

Das Paradigma der Service-Orientierung beschreibt wie erwähnt zunächst lediglich ein Prinzip, nach welchem eine Aufgabe gelöst wird. Man benötigt eine konkrete Implementierung einer SOA, die dieses Paradigma realisiert. Mittels Technologien wie die bereits erwähnte CORBA (*Common Object Request Broker Architecture*) und *Web Services* kann eine SOA umgesetzt werden.[3] Im Folgenden wird auf die Technologie der Web Services beispielhaft eingegangen.

[3] Bei dieser getroffenen Aussage herrscht Uneinigkeit, inwieweit mit Web Services alle Anforderungen einer SOA erfüllt werden können. Diese Diskussion wird jedoch nicht im Rahmen dieser Arbeit geführt.

Ähnlich wie bei der SOA gibt es auch für Web Services verschiedene Definitionen. An dieser Stelle wird die Definition des W3C (*World Wide Web Consortiums*) [119] für Web Services aufgeführt. Das W3C wurde von Tim Berners-Lee, dem Begründer des WWW, mit Unterstützung des europäischen Kernforschungszentrum CERN, der DARPA und der Europäischen Union ins Leben gerufen, mit dem Ziel, Protokolle und Richtlinien für das WWW zu entwickeln und zu standardisieren. Das W3C definiert einen Web Service als: *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.* [124] Aus dieser Definition sind die Ziele der Web Service Technologie deutlich herauszulesen. So wird hier zum einen die Maschine-zu-Maschine-Kommunikation betont, welche eine Schnittstellenbeschreibung in maschinenlesbarer Form bedingt. Zum anderen wird die Kommunikation über ein Netz mittels im WWW standardisierter Technologien hervorgehoben.

Die bei Web Services verwendeten Technologien sowie deren Zuordnung zu den einzelnen Aufgaben sind in Abbildung 2.9 vom W3C zusammengefasst.

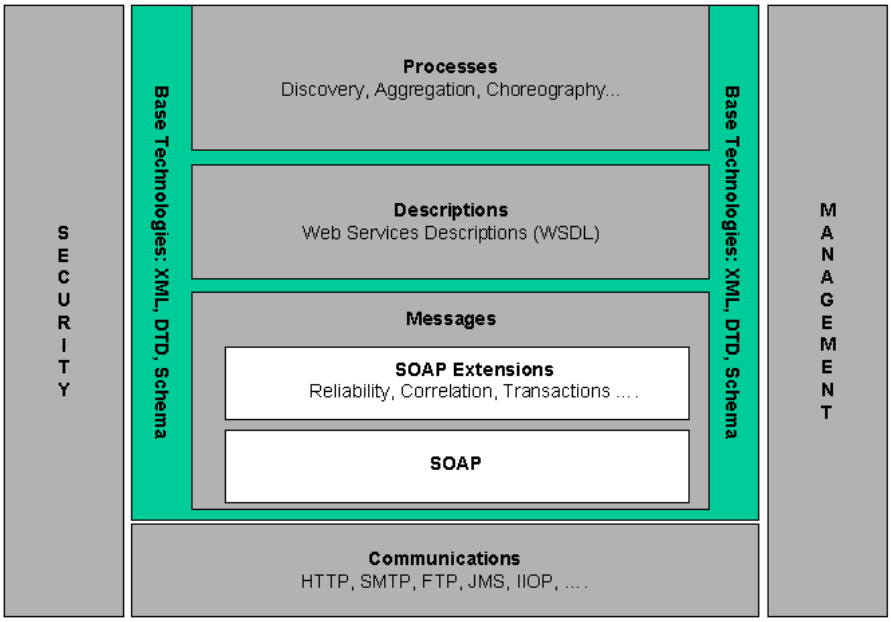


Abbildung 2.9: Web Service Technologien in der Übersicht (Quelle: [124])

Die Kommunkationsgrundlage, in Abbildung 2.9 als Schicht *Communications* dargestellt, bilden dabei verschiedene bereits standardisierte Internetprotokolle wie HTTP (*Hypertext Transfer Protocol*) [21], SMTP (*Simple Mail Transfer Protocol*) [47] oder FTP (*File Transfer Protocol*) [87]. Wie die vom W3C angegebene Definition besagt, ist die Verwen-

dung solcher standardisierter Protokolle typischerweise erwünscht aber nicht zwingend gefordert.

Oberhalb der Kommunikationsschicht befindet sich die Nachrichtenschicht (*Messages*). Hier wird das Protokoll SOAP (*kein Akronym*) [121][122] verwendet. SOAP basiert auf der Auszeichnungssprache XML (*Extensible Markup Language*) [123]. Durch die Verwendung von XML wird mittels SOAP eine plattformunabhängige und sowohl menschen- als auch maschinenlesbare Repräsentation der Daten realisiert.

Die über *Messages* liegende Schicht *Descriptions* beinhaltet die Beschreibung der Web Service Schnittstellen. Mit der ebenfalls XML-basierten Beschreibungssprache WSDL (*Web Services Description Language*) [120] können die Schnittstellen mit ihren Operationen sowie die technischen Informationen der Verbindungsendpunkte beschrieben werden. Durch die abstrakte Beschreibung der Schnittstelle kann ein entfernter Dienst ohne Wissen über Implementierungsdetails aufgerufen und genutzt werden. Dies ermöglicht die flexible Komposition von Anwendungen aus unterschiedlichen Diensten.

Die oberste Schicht *Processes* behandelt alle Aktionen in Bezug auf Web Services, wie beispielsweise das Auffinden (engl.: *discovery*) von Ressourcen bzw. weiteren Web Services im Netz. Das Auffinden wird über einen sogenannten Verzeichnisdienst realisiert.

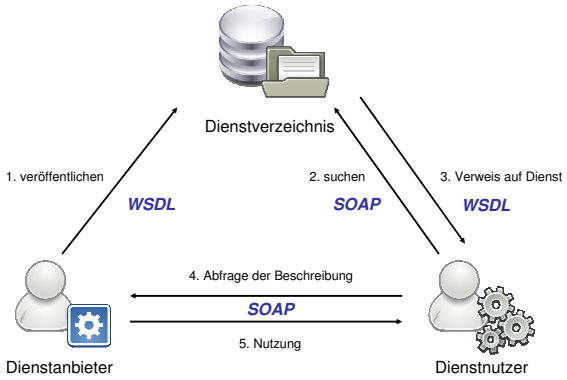


Abbildung 2.10: Web Service Technologien beim Zusammenspiel zwischen den einzelnen Rollen einer SOA

Das Zusammenspiel der vom W3C festgelegten Web Service Techniken zur Realisierung einer SOA und der in Abbildung 2.8 aufgeführten Rollen ist in Abbildung 2.10 dargestellt. Lediglich über die Realisierung des Verzeichnisdienstes werden seitens des W3C keine genauen Aussagen zur zu verwendenden Technik gemacht.

Mittels Web Services lassen sich somit Dienste erstellen und Anwendungen unter Verwendung dieser Dienste flexibel orchestrieren.

3 Pacemates - Sensorknoten für praktische Anwendungen

Ziel der Arbeit ist es, ein Paradigma und Algorithmen für praxistaugliche Sensornetze zu erarbeiten und umzusetzen. Wie bereits in der Einleitung motiviert werden insbesondere Algorithmen und Protokolle meist nur mittels Simulationen evaluiert. Die Bedeutung der Ergebnisse ist wie beschrieben nicht immer praxisrelevant. Um also Praxistauglichkeit und somit den wirklichen Wert von Paradigmen und Algorithmen aufzuzeigen, müssen sich diese in realen Netzen beweisen. Dazu wird eine Plattform benötigt, mittels derer man die Paradigmen und Algorithmen anwenden und evaluieren kann. Im Rahmen dieser Arbeit sind für das MarathonNet-Forschungsprojekt die *Pacemate*-Sensorknoten [63] vom Autor entwickelt worden, die in diesem Kapitel vorgestellt werden.

Im Folgenden wird zunächst auf bestehende Realisierungen von Sensorknoten eingegangen. Danach werden die Anforderungen formuliert, die sich aus dem MarathonNet-Anwendungsszenario ergeben. Hierbei werden zunächst praktische Anforderungen formuliert, die die „äußere“ Beschaffenheit des Sensorknotens betreffen. Anschließend werden die netzspezifischen Anforderungen im Hinblick auf das Anwendungsszenario analysiert und evaluiert. Neben den anwendungsspezifischen Anforderungen werden außerdem die Anforderungen, die sich aus der Perspektive der Forschung ergeben, aufgeführt. Anschließend werden das anhand dieser Anforderungen realisierte Hardwaredesign sowie die Softwareschnittstellen zur Hardware vorgestellt.

3.1 Verwandte Arbeiten

Der Begriff des Sensorknotens wurde bereits in Kapitel 2 eingeführt. Dabei wurden auch die Grundkomponenten der Hardware beschrieben: Prozessor, Speicher, Funkschnittstelle und Sensor. Es haben sich gerade im Rahmen der aktiven Forschung im Bereich der Sensornetztechnologie viele verschiedene Hardwareplattformen entwickelt. Im Folgenden wird exemplarisch auf zwei etablierten Plattformen *MICA* und *BTnodes* eingegangen.

3.1.1 MICA

Die am weitesten verbeitete Plattform für Sensorknoten im wissenschaftlichen Bereich ist der MICA Knoten [35][36] von der Firma Crossbow Technology Incorporated [11]. Ursprünglich wurde diese Plattform an der University of California in Berkeley entwickelt. Aufbauend auf dem Forschungsprototypen dem Rene Sensorknoten [37] von

1999 und der Entwicklung des Sensorknotenbetriebssystems TinyOS[1] wurde anhand der gewonnenen Erfahrungen und Erkenntnisse die MICA Sensorknotenplattform entworfen. In 2001 übernahm die Firma Crossbow Technology Incorporated den Vertrieb der MICA-Knoten und wurde somit zum ersten kommerziellen Anbieter von drahtlosen Sensormodulen. In 2002 belieferte Crossbows Technology die Forschungseinrichtung des amerikanischen Militärs DARPA (Defense Advanced Research Projects Agency) [13] im Rahmen des SenseIT Programms, um den bis dahin größten Feldtest für Sensornetze durchzuführen. Crossbow Technology vertreibt zum aktuellen Zeitpunkt zwei Versionen der MICA-Reihe: MICA2 und MICAz. Dabei ist der MICA2 die Weiterentwicklung des ursprünglichen MICA-Knotens. Tabelle 3.1 gibt eine Übersicht über die technischen Daten der drei Sensorknoten.

		MICA	MICA2	MICAz
Microkontroller	Fabrikat	ATMEL ATmega 103L	ATMEL ATmega 128	ATMEL ATmega 128
	Architektur	8 bit AVR	8 bit AVR	8 bit AVR
Speicher	Programmspeicher (Flash)	128 kbyte	128 kbyte	128 kbyte
	Externer Speicher (Serial Flash)	-	512 kbyte	512 kbyte
	Konfigurations-speicher (EEPROM)	4 kbyte	4 kbyte	4 kbyte
	RAM	4 kbyte	4 kbyte	4 kbyte
Funkmodul	Fabrikat	RF Monolithics TR1000	Chipcon CC1000	Chipcon CC2420
	Sendefrequenz	916 MHz	868/916 MHz	2,4 GHz
	Datenrate	13,3 kbit/s	19,2 kbit/s	256 kbit/s
	max. Sendeleistung	1,5 dBm	10 dBm	0 dBm
Versorgung	Spannung	3 V	3 V	3 V
	Batterien	2 × AA	2 × AA	2 × AA

Tabelle 3.1: Technische Daten der Sensorknotenplattformen MICA, MICA2 und MICAz

Im Bereich der eingebetteten Systeme werden Mikrocontroller verwendet. Mikrocontroller stellen sozusagen ein *System on a Chip* dar, indem sie zusätzlich zum Prozessor auch Peripheriefunktionalität in sich vereinen. Dazu gehört beispielsweise der Programmspeicher sowie serielle Schnittstellen.In Tabelle 3.1 ist zu erkennen, dass sich die Leistungsdaten des MICA2 gegenüber dem ursprünglichen Knoten nur unwesentlich verändert haben. Als Hauptveränderung ist die Ergänzung eines externen Datenspeichers zu nennen. Da Sensorknoten für das Aufzeichnen von Daten verwendet werden, ist es gerade

[1]Das Betriebssystem TinyOS wird im Kapitel 4 vorgetellt.

im Forschungsbereich sinnvoll, die Daten nicht nur direkt durch das Netz weiterzuleiten sondern auch lokal persistent vorzuhalten. Dadurch lässt sich die Evaluation von verschiedenen Verfahren vereinfachen, da man zusätzlich Statistiken auf dem Knoten speichern kann und gegebenenfalls genauere Aussagen über beispielsweise Paketverluste treffen kann. Hierbei ist der externe Datenspeicher, der bei MICA2 und MICAz hinzugefügt wurde, von Vorteil.

Die zweite Veränderung liegt im Funkmodul. Das für den MICA verwendete Funkmodul TR1000 von RF Monolithics benutzt Amplitudenumtastung (engl.: *Amplitude Shift Keying* (ASK)) zur Modulation des Funksignals und erreicht eine Datenübertragungsrate von 13,3 kbit/s. Das Funkmodul CC1000 von Chipcon des MICA2-Knoten verwendet Frequenzumtastung (engl.: *Frequency Shift Keying* (FSK)) und erreicht eine Datenübertragungsrate von 19,2 kbit/s. Beide Module nutzen die Frequenz von 919 MHz und liegen somit für Nord- und Südamerika in einem ISM-Band (Industrial, Scientific and Medical Band), einem Frequenzbereich von 902 bis 928 MHz, der von Geräten in Industrie, Wissenschaft und Medizin frei genutzt werden kann. Das CC1000 Modul des MICA2 kann zusätzlich im Bereich vom 868 MHz betrieben werden, welches innerhalb Deutschlands für Funkanwendungen ohne Bedarfsnachweis genutzt werden kann. In dem MICAz ist das Modul CC2420 der Firma Chipcon verwendet. Das Modul nutzt die Frequenz von 2,4 GHz und erreicht eine Datenrate von 256 kbit/s. Zudem ist es kompatibel zum IEEE 802.15.4 Standard für Übertragungsprotokolle in Wireless Personal Area Networks (WPAN) und erlaubt somit eine einfache Integration in andere Netze und den darin laufenden Applikationen.

Die MICA Sensorknoten kamen bei der in [71] genannten ersten Realisierung zur Überwachung des Brutverhaltens von Seevögeln auf Great Duck Island zum Einsatz. Die zweite Realisierung auf Great Duck Island [99] benutzte bereits die MICA2 Knoten, die für das Szenario in ihrer Größe und Form speziell angepasst wurden. Auch für das genannte Projekt Code Blue [68] wurden bereits die MICA2 Knoten verwendet.

3.1.2 BTnodes

Drei Jahre nach der Vorstellung der MICA Plattform wurden 2003 die BTnodes [4] von der Eidgenössischen Technischen Hochschule (ETH) Zürich vorgestellt. Ziel war es, Sensornetzanwendungen schnell und einfach in Alltagsanwendungen zu integrieren und die proprietären Kommunikationsbegrenzungen aufzuheben, indem man auf das standardisierte Übertragungsprotokoll für mobile Kleinstgeräte Bluetooth (gemäß IEEE 812.15.1) zurückgreift. Mittlerweile gibt es die dritte Generation von BTnodes [6], welche auch gewerblich vertrieben wird. Als Neuerung kam neben einem verbesserten Bluetooth Modul insbesondere ein zweites Funkmodul hinzu. Die BTNodes verwenden wie in Tabelle 3.2 dargestellt parallel zu dem auf 2,4 GHz operierenden Bluetooth Modul das Modul CC1000 der Firma Chipcon (identisch zu MICA2), um nicht auf Bluetooth Kommunikation beschränkt zu sein. Die BTnodes haben ihr eigenes Betriebssystem BTnut. Seit der zweiten Generation existiert jedoch auch eine Portierung des von der MICA-Serie

verwendeten TinyOS Betriebssystems, welche auf der BTnode Internetpräsenz der ETH Zürich zur Verfügung steht.

		BTnode rev3	
Microkontroller	Fabrikat	ATMEL ATmega 128L	
	Architektur	8 bit AVR	
Speicher	Programmspeicher (Flash)	128 kbyte	
	Externer Speicher (Serial Flash)	-	
	Konfigurations-speicher (EEPROM)	4 kbyte	
	RAM	64+180 kbyte	
Funkmodul	Fabrikat	Zeevo ZV4002	Chipcon CC1000
	Sendefrequenz	2,4 GHz	868/916 MHz
	Datenrate	256 kbit/s	19,2 kbit/s
	max. Sendeleistung	4 dBm	10 dBm
Versorgung	Spannung	3,8 - 5 V	
	Batterien	2 × AA	

Tabelle 3.2: Technische Daten der Sensorknotenplattform BTnode

3.2 Anforderungen

In den vorangegangenen Kapiteln wurde bereits auf die vielfältigen Einsatzmöglichkeiten der Sensornetztechnologie eingegangen. Insbesondere wurde in Kapitel 2.1 auf die unterschiedlichen Anforderungen und Charakteristika verschiedener Anwendungsszenarien eingegangen. Diese wirken sich auch auf die Anforderungen an die verwendete Hardware aus. Wie der vorangegangene Abschnitt gezeigt hat, können sich die verschiedenen Plattformen derart unterscheiden, dass sie sich für ein zugrundeliegendes Szenario besser oder schlechter eignen. Beispielsweise verfügen die BTnodes lediglich über 128 kbyte Flash Speicher. Dies bedeutet, dass neben der eigentlichen Anwendung, die persistent im Flash gespeichert wird, nur noch wenig Speicher für persistent zu haltende Daten, also Daten, die auch nach einen Neustart des Gerätes vorhanden sind, zur Verfügung steht. Die zuvor aufgeführten MICA2 und MICAz verfügen dem gegenüber neben den 128 kbyte Programmspeicher über den zusätzlichen externen Speicher von 512 kbyte auf dem Daten persistent gespeichert werden können. Die Anforderungen an die Funkeigenschaften müssen ebenfalls berücksichtigt werden. So ermöglichen die Funkmodule

mit der Sendefrequenz von 2,4 GHz eine wesentlich höhere Datenrate. Diese Eigenschaft kann für bestimmte Anwendungen unabdingbar sein. Auch weitere Eigenschaften, wie Funkreichweite, Formfaktor, Lebensdauer (bzgl. der Energieversorgung), Robustheit gegenüber mechanischen Einwirkungen sowie Staub und Wasser können für verschiedene Anwendungsszenarien von unterschiedlicher Wichtigkeit sein. In diesem Abschnitt werden daher die Anforderungen an die in dieser Arbeit entwickelten Sensorknotenplattform genauer betrachtet. Dabei werden anhand von simulativen Untersuchungen die technischen Anforderungen an die Plattform spezifiziert.

3.2.1 Anwendungsspezifische Anforderungen

Im Rahmen des während dieser Arbeit durchgeführten MarathonNet-Projektes wurden praktische Erfahrungen mit mobilen Sensornetzen gesammelt. Exemplarisch sind dazu Sensornetze bei Laufveranstaltungen eingesetzt worden. Die erstellte Applikation zeichnet dabei biometrische Daten der Läufer über den Rennverlauf auf. Diese Daten stehen dann während der Laufveranstaltung sowohl dem Läufer selbst, dem Betreuer, aber auch dem Organisator und dem Publikum zur Verfügung. Eine Hauptanforderung bestand darin, dass das Sensornetz jederzeit Position und Platzierung sowie die aktuelle Geschwindigkeit des Läufers verfügbar macht. Diese Daten stehen dann während der Laufveranstaltung sowohl dem Läufer selbst, dem Betreuer, aber auch dem Organisator und dem Publikum zur Verfügung. Im Rahmen des MarathonNet-Projektes sind dabei 500 Sensorknoten produziert worden.

Praktische Anforderungen

Aus diesem Anwendungsszenario ergeben sich direkt Anforderungen an die verwendete Sensornetzplattform. Um biometrische Daten eines Läufers zu erfassen, muss der Läufer den Sensorknoten beim Lauf mit sich tragen. Dies bedeutet, dass der Sensorknoten den hohen ergonomischen Anforderungen der Athleten entsprechen muss. Der Sensorknoten darf nicht zu schwer sein und sollte sich angenehm und für den Läufer gut sichtbar tragen lassen. Außerdem muss er gegen die bei einer Laufveranstaltung entstehenden Umwelteinflüsse wie physikalische Belastung, Nässe, Schweiß und Staub geschützt sein. Dies bedeutet, dass der Sensorknoten in einem ergonomischen und gleichzeitig robusten Gehäuse unterzubringen ist. Um dem Läufer die beschriebenen Dienste wie Anzeige der Position und Platzierung zu bieten, muss der Sensorknoten über eine Anzeige und Bedienungselemente wie Tasten verfügen. Der Sensorknoten muss über eine entsprechende Sensorik verfügen, die biometrische Daten des Läufers messen kann. Diese Eigenschaften sind mit denen einer herkömmlichen Pulsuhr mit großen Display vergleichbar.

Netzspezifische Anforderungen

Im Rahmen der Bestimmung der Anforderungen an die Sensorknoten müssen Untersuchungen über die zu erwartende Verteilung von Läufern bei einer Laufveranstaltung

durchgeführt werden. Hierzu ist es notwendig anhand eines Modells die Anforderungen der Anwendung an das zu realisierende Netz zu spezifizieren, um daraus die Anforderungen an die Sensorknoten ableiten zu können.

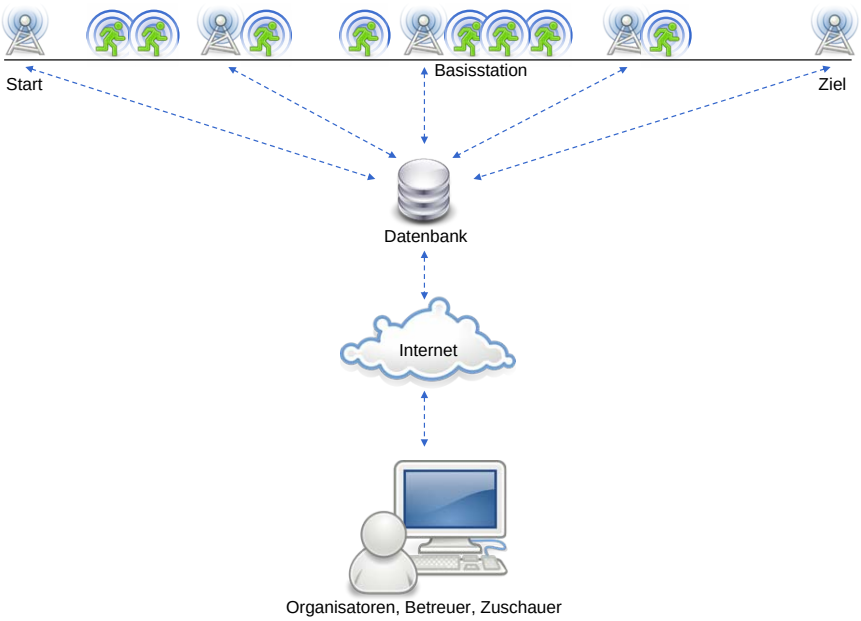


Abbildung 3.1: Aufbau des MarathonNet-Systems

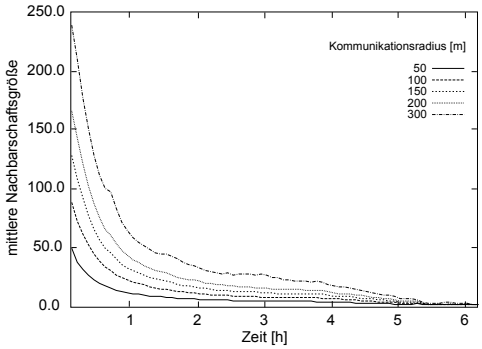
In Abbildung 3.1 ist der zu realisierende Aufbau des MarathonNet-Systems dargestellt. Um jederzeit Daten einzelner Läufer aus dem entstehenden Sensornetz zu erhalten, muss das Netz permanent mit sogenannten *Gateways* bzw. *Basisstationen* in Verbindung stehen, die entlang der Strecke verteilt sind. Diese *Basisstationen* stellen die Verbindung zwischen dem Sensornetz und einer zentralen Datenbank dar, in der die gemessenen Daten abgelegt werden, um anschließend aufbereitet dargestellt werden zu können. Zum einen muss die Anzahl solcher *Basisstationen* bestimmt werden. Zum anderen muss jedoch insbesondere die benötigte optimale Kommunikationsreichweite der Sensorknoten bestimmt werden. Diese beiden Parameter müssen so gewählt werden, dass möglichst über den gesamten Rennverlauf für jeden Sensorknoten direkt oder multi-hop über andere Knoten einen Pfad zu einer solchen *Basisstation* existiert. Weiterhin muss die benötigte Bandbreite auf dem Funkmedium abgeschätzt werden, so dass alle zu erhebenden Daten der 500 Läufer über das Funkmedium übertragen werden können. Im Folgenden werden die Messungen und Simulationen beschrieben anhand derer die Anzahl der Basisstationen, die benötigte Funkreichweite der Sensorknoten und die Datenrate bestimmt wurden. Diese Evaluation wurde in [32][85] veröffentlicht.

Zur Analyse dieser netzspezifischen Anforderungen wird dabei auf den Netzsimulator *Shawn* [20] zurückgegriffen. Der Simulator wurde im Rahmen dieser Arbeit dahingehend

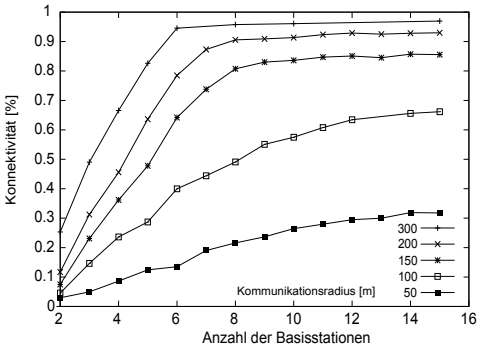
(a) 00:36:40 (b) 01:33:20 (c) 02:03:20 (d) 02:55:00

Abbildung 3.2: Läuferpositionen zu bestimmten Zeitpunkten des Rennens (hh:mm:ss)

erweitert, dass das Marathonszenario in *Shawn* abgebildet wird. Da es von Marathonveranstaltungen öffentlich verfügbare Zwischenzeiten gibt, kann auf diese Aufzeichnungen als Eingabe zur Bewegung der Läufer zurckgegriffen werden. Die Anzahl der Sensorknoten wird dabei auf 500 Geräte festgelegt, so dass eine Auswahl von Läufern mit diesen Knoten ausgestattet wird. Bei der Bestimmung der Kommunikationsreichweite muss die einer Laufveranstaltung zugrundeliegende Netzdynamik berücksichtigt werden. Wie Abbildung 3.2 anhand der Zwischenzeiten und des Streckenverlaufes des Hamburg Marathons im Jahr 2005 zeigt, kann sich ein Feld von Läufern im Verlauf eines Marathonlaufes bis über die Hälfte der Strecke ausdehnen. Sind die als Kreuze dargestellten Läufer zu Beginn des Rennens (Abbildung 3.2(a)) noch dicht beisammen, so dehnt sich das Feld mit fortschreitender Zeit (Abbildung 3.2(b) und Abbildung 3.2(c)) immer weiter aus. Zum Zeitpunkt bei 2 Stunden und 55 Minuten in Abbildung 3.2(d) erkennt man, dass die letzen Läufer erst ungefähr die Hälfte der Strecke absolviert haben, wobei die ersten sich schon lange im Ziel befinden. Diese Entwicklung bedeutet insbesondere, dass sich die Dichte des Feldes und somit auch die Dichte des aus den Sensorknoten bestehenden Netzes über den Rennverlauf ändert. Eine geringere Dichte bedeutet dabei, dass jeder Knoten im Durchschnitt weniger direkte (1-hop) Nachbarn besitzt. Abbildung 3.3(a) zeigt dabei den durchschnittlichen Verlauf der 1-hop-Nachbarschaftsgrößen für 500 Läufer bei verschiedenen zugrundeliegenden Kommunikationsradien von 50 m bis 300 m. Die starke Ausdehnung des Feldes spiegelt sich im starken Abfall der Nachbarschaftsgrößen wieder. Abbildung 3.3(b) zeigt für unterschiedliche Kommunikationsradien den durchschnittlichen Prozentsatz der Zeit, für den die einzelnen Läufer einen Pfad zu mindestens einer Basisstation haben. Die oben beschriebenen Dienste des MarathonNet-Szenarios wie besipielsweise Bestimmung der Position und Übertragung der Position von einem Mitläufer tolerieren kurze Zeiten der Diskonnektivität, in denen das System extrapolieren kann. Unterhalb einer Konnektivität von 90 % der Renndauer wird eine solche Funktionalität, wie z.B. den Abstand zu den Laufpartnern anzuzeigen, zu ungenau, da die Geschwindigkeiten der Läufer ständigen Änderungen unterliegen. Man erkennt deutlich, dass es eine Mindestanzahl von 6-10 Basisstationen gibt, die abhängig vom Kommunikationsradius für diese Konnektivität notwendig ist. Eine weitere Steigerung der Anzahl der Basisstationen bringt für die Konnektivität keine Verbesserung. Eine Erhöhung des Kommunikationsradius hingegen kann die Konnektivität weiter verbessern. Aus Abbildung 3.3 lässt sich als Anforderung an die Sensorknoten ein Kommunikationsradius von mindestens 200 m herauslesen.



(a) Verlauf der mittlere Nachbarschaftsgrößen über das Rennen



(b) Mittlerer Prozentsatz der Zeit, in der ein Pfad zwischen Sensorknoten und Basisstation existiert

Abbildung 3.3: Simulative Evaluation der Nachbarschaften und der Verbindung zu Basisstationen für 500 Läufer

Wert	Intervall / Genauigkeit	Größe
Läufer Id	0-65535 / 1	2 byte
Zeitstempel	0-6 h (0-21600 s) / 1 s	2 byte
Position	0-42000 m / 1 m	2 byte
Biometrische Daten (beispielsweise Herzfrequenz)	0-240 / 1	1 byte

Tabelle 3.3: Zu versendende Datentupel im MarathonNet-Szenario

Abschließend ist das zu erwartende Datenaufkommen im MarathonNet-Szenario abzuschätzen, um eine Aussage über die benötigte Übertragungsrate der Funkschnittstelle der Sensorknoten zu bekommen. Wie oben bereits beschrieben werden regelmäßig biometrische Daten und Positionen erfasst. Diese werden zusammen mit einem Zeitstempel und der Läuferkennung über die Basisstationen an die Datenbank übertragen. Diese in Tabelle 3.3 dargestellten Datenpakete der Größe $d = 7$ byte werden in einem Intervall $t = 60$ s versandt. Weiterhin kann man der Berechnung pessimistische zusätzliche Kommunikationskosten (*Overhead*) von $o = 70$ byte zu Grunde legen, die sich aus den verwendeten Kommunikationsprotokollen ergeben. Dieser Wert ist vergleichbar mit dem Overhead bei IP oder Ethernet. Daraus ergibt sich als Anforderung für die Bandbreite b_s eines einzelnen Knotens:

$$b_s = \frac{d+o}{t} = \frac{(7 \text{ byte } + \ 70 \text{ byte}) * 8 \text{ bit/byte}}{60 \text{ s}} = 10 \text{ bit/s} \tag{3.1}$$

Untersuchungen aus [59] haben ergeben, dass bei einer MAC-Realisierung gemäß 802.11, welche RTS, CTS und ACK verwendet, nur $p = 1/7$ der verfügbaren Bandbreite effektiv genutzt werden kann. Wieder unter der Annahme des schlechtesten Falls, bei dem ein

einzelner Knoten alle Pakete der anderen Läufer versenden muss, ergibt sich bei $N = 500$ Läufern als Anforderung der Bandbreite b an das Funkmedium:

$$b = \frac{b_s * N}{p} = \frac{10 \text{ bit/s} * 500}{\frac{1}{7}} = 35 \text{ kbit/s} \tag{3.2}$$

3.2.2 Forschungsspezifische Anforderungen

Da das MarathonNet-Projekt ein Forschungsprojekt ist, darf nicht nur die Anwendung im Vordergrund stehen. Ein Ziel des Forschungsprojektes ist es, den Transfer von simulativ untersuchten Protokollen und Algorithmen zur realen Umsetzung zu schaffen. Dies bedeutet, dass das Anwendungsszenario die Basis für experimentelle Untersuchungen bildet. In dieser Hinsicht ergeben sich weitere Anforderungen an die Sensorknotenplattform.

Da die Geräte im täglichen Forschungsbetrieb genutzt werden und insbesondere sowohl bei Experimenten wie auch beim MarathonNet-Szenario bei einer Laufveranstaltung in großen Stückzahlen (eben bis zu 500 Stück) zum Einsatz kommen, müssen die Batterien der Geräte auf einfache Weise aufgeladen werden können. Ein ständiges manuelles Auswechseln der Batterien ist bei dieser Anzahl unmöglich. Auch widerspricht ein dafür benötigtes Batteriefach der Anforderung, ein gegen Staub und Nässe geschütztes Gehäuse zu haben. Ein Batteriefach, welches diesen Anforderungen entspricht, in das Gerät zu integrieren, wäre dabei zu kostenintesiv. Es wird eine Lösung benötigt, die ein einfaches simultanes Aufladen mehrerer Geräte ermöglicht. Dabei sollen insbesondere keine speziellen Ladegeräte entwickelt werden, sondern es muss auf verfügbare Standardladegräte zurückgegriffen werden können.

Im Zentrum der Forschung stehen Sensornetzanwendungen und somit verteilte Anwendungen, deren Hauptcharakteristik das Zusammenwirken und die Kommunikation zwischen mehreren Geräten ist. Um das korrekte Verhalten in der Praxis zu prüfen und insbesondere zu evaluieren, müssen die Anwendungen und Algorithmen auf mehrere Geräte gleichzeitig gespielt werden. Da gerade im Forschungsbetrieb immer wieder kleinere Änderungen an den Programmen vorgenommen werden, muss der Vorgang des Überspielens der Anwendungen auf die Sensorknoten so wenig Zeit wie möglich in Anspruch nehmen. Deshalb ist eine Programmierung über Funk unbedingt notwendig. Dies macht manuelles Verbinden von Programmieradaptern an die Geräte überflüssig. Gleichzeitig ermöglicht die Tatsache, dass Funknachrichten von jedem Sensorknoten gehört werden können, ein paralleles Programmieren mehrerer Geräte.

Um forschungsrelevante Ergebnisse zu produzieren, müssen etliche Daten aufgezeichnet und später ausgewertet werden. Zudem bietet es sich an, in einem experimentellen Aufbau die zu vergleichenden Protokolle und Algorithmen zeitgleich auf den Geräten vorzuhalten. Ein Umstellen von einem Algorithmus oder Protokoll zum anderen ist zum einen einfacher, zum anderen kann man so leichter garantieren, dass beide Algorithmen unter gleichen Bedingungen getestet werden, da Zeit, Platzierung der Geräte sowie die Geräte selbst die gleichen sind. Aus der Notwendigkeit, viele Daten zu erheben sowie

verschiedene Algorithmen und Protokolle auf den Geräten zu haben, ergibt sich die Anforderung, dass die Geräte über einen ausreichend großen Speicher verfügen müssen. Insbesondere darf eine Firmware, die den Zugriff auf Hardwareschnittstellen wie Funk oder Display realisiert, nur einen geringen Teil des Speichers belegen.

Mittels eines Sensornetzes sollen Daten erhoben werden, die dann außerhalb des Netzes zur Verfügung stehen. Insbesondere im Rahmen der Experimente mit den Sensorknoten muss es möglich sein, Messdaten bzw. erhobene Statistiken sogar von einzelnen Geräten auslesen zu können. Hierzu ist es notwendig, die Sensorknoten an einen externen Rechner anschließen zu können. Eine weitere Anforderung ist daher, dass eine Kommunikation vom Sensorknoten zum Rechner wie auch vom Rechner zu einem Sensorknoten möglich sein muss.

3.3 Hardwaredesign

Basierend auf den oben beschriebenen Anforderungen ist die Sensorknotenplattform *Pacemate* entwickelt worden. Die *Pacemates* wurden als Kompromiss zwischen dem technisch Machbaren und dem engen Finanzrahmen des MarathonNet-Projektes entwickelt. Ein Einsatz existierender Plattformen wie beispielsweise der in Kapitel 3.1 aufgeführten Knoten schied aufgrund der Ergonomie und den speziellen durch das MarathonNet-Szenario gegebenen Anforderungen wie beispielsweise einer großen Anzeige aus.

Es wird schnell klar, dass man bei dem *Pacemate*-Sensorknoten die Form- und Größeneigenschaften ähnlich einer Pulsuhr nicht realisieren kann. Dies ist zum einen den besonderen Anforderungen zum anderen aber auch dem zur Verfügung stehenden finanziellen Rahmen geschuldet. Um dennoch den ergonomischen Anforderungen Genüge zu tun, wurde ein Gehäusedesign entwickelt, das sich auf dem Handrücken tragen lässt. Abbildung 3.4 zeigt den Entwicklungsprozess des *Pacemate* von der ersten Idee über die Studie und Konstruktion bis hin zum fertigen Kleinseriengerät. Das Design wurde dabei

(a) Idee (b) Studie (c) Konstruktion (d) Serie

Abbildung 3.4: Entwicklung des *Pacemate* von der Idee bis zum fertigen Kleinseriengerät

basierend auf ergonomischen Richtlinien [93] entwickelt.[2] Die Anzeige mit der Auflösung 128×64 ist für den Athleten gut sichtbar und erlaubt durch die Größe die Darstellung

[2]Die abgebildete Idee und Studie wurde in Zusammenarbeit mit SBm mediengrafik entworfen.

verschiedenster Informationen und Grafikelemente, was sowohl für komplexere Anwendungen wie auch für die Anzeige experimenteller Daten zu Forschungszwecken von Nutzen ist. Mittels der drei leicht ertastbaren Knöpfe kann der Benutzer mit dem *Pacemate* interagieren. Das Gehäuse ist gemäß Schutzklasse IP56 (*ingress protection*) konstruiert und ist somit gegen Staub und Strahlwasser geschützt.

Als Sensor verfügt der *Pacemate* über ein RMCM-01 Heart Rate Receiver Chip der Firma Polar [86] zum Empfang von Pulsdaten eines kodierten Polar-Pulsgurtes. Die Entscheidung zu dieser Schnittstelle wurde auf Basis einer durchgeführten Umfrage [34] getroffen, wonach die Mehrheit der Athleten bereits über einen Pulsmesser der Firma Polar oder ein kompatibles Gerät verfügt.

Der *Pacemate* verfügt über ein Funkmodul XE1205 der Firma Semtech, welches sich in den Frequenzbändern 433 MHz, 868 MHz und 915 MHz betreiben lässt. Seitens der Firmware wird das Funkmodul jedoch standardmäßig auf dem in Deutschland ohne Bedarfsnachweise nutzbaren Frequenzband von 868 MHz betrieben. Das Modul erreicht eine Datenrate von 152 kbit/s bei einer Sendeleistung von 15 dBm und erfüllt somit die beschriebenen Anforderungen an Reichweite und Datenrate.[3]

Als Mikrocontroller wird im *Pacemate* der Philips LPC2136 verwendet. Dieser Mikrocontroller wird standardmäßig in einem 32 bit Modus betrieben, was bedeutet, dass jede Instruktion (ggf. mit Argumenten) des Maschinenbefehlsatzes durch einen 32 bit breiten Zahlencode dargestellt wird. Der LPC2136 verfügt über 32 kbyte RAM und 256 kbyte Programmspeicher. Tabelle 3.4 zeigt die technischen Daten des *Pacemates* nochmals in der Übersicht. Im Gegensatz zu den zuvor aufgeführten MICA Plattformen verfügt der *Pacemate* somit über einen achtmal größeren RAM, was ein nichtpersistentes Zwischenspeichern von Daten (beispielsweise: Pulsdaten der Läufer) ermöglicht. Im Vergleich zu den MICA Knoten und dem BTNode verfügt der *Pacemate* über einen doppelt so großen Programmspeicher, verzichtet aber dafür auf zusätzlichen externen Speicher.

Der Speicherplan des LPC2136 ist in Abbildung 3.5 dargestellt. Die 32 kbyte RAM liegen dabei im Addressbereich von 0x4000 0000 bis 0x4000 7FFF. Der Programmspeicher (hier Flash) liegt im Bereich von 0x0000 0000 bis 0x0003 FFFF. Der Programmspeicher ist dabei in 15 Sektoren unterteilt. Dabei sind die Sektoren 0 bis 7 jeweils 4 kbyte groß und die Sektoren 8 bis 14 haben eine Größe von 32 kbyte. Der LPC2136 Mikrocontroller erlaubt ein Löschen und Beschreiben des persistenten Programmspeichers durch die laufende Applikation. Der persistente Programmspeicher lässt sich jedoch nicht einfach überschreiben, wie beispielsweise der RAM, auf den lesend und schreibend zugregriffen werden kann. In einem leeren Programmspeicher ist jedes Bit auf den Initialwert 1 gesetzt. Beim Beschreiben können lediglich Einsen in Nullen umgewandelt werden. Da es sich bei dem vorliegenden Programmspeicher um einen Flash-EEPROM (*Electrically Erasable Programmable Read-Only Memory*) handelt, kann dieser jedoch sektorweise wieder zurückgesetzt (gelöscht) werden. Das Löschen eines Sektors oder des gesamten

[3] Die Reichweite wurde in mehren Tests verifiziert und unterliegt Schwankungen hervorgerufen durch Umwelteinflüssen, wie beispielsweise Dämpfung durch Gebäude, Personen u.ä..

		Pacemate
Microkontroller	Fabrikat	Philips LPC2136
	Architektur	32 bit ARM
Speicher	Programmspeicher (Flash)	256 kbyte
	Externer Speicher (Serial Flash)	-
	Konfigurations-speicher (EEPROM)	-
	RAM	32 kbyte
Funkmodul	Fabrikat	Semtech XE1205
	Sendefrequenz	868 MHz
	Datenrate	152 kbit/s
	max. Sendeleistung	15 dBm
Versorgung	Spannung	1,5 V
	Batterien	1 × AA

Tabelle 3.4: Technische Daten der *Pacemate*-Sensorknotenplattform

Programmspeichers benötigt dabei 400 ms. Das Schreiben von 256 byte in diesen Programmspeicher benötigt 1 ms.

Der *Pacemate* wird durch einen normalen Nickel-Metallhydrid-Akkumulator vom Typ AA versorgt. Der verwendete Akkumulator Sanyo HR-3U hat dabei eine typische Kapazität von 2500 mAh bei einer Spannung von 1,2 V. Da die Elektronik des *Pacemates* mit 3 V betrieben werden muss, ist ein Spannungswandler in die Schaltung integriert, der die 1,2 V des Akkumulators auf 3 V hochreguliert. Die Pole des Akkumulators sind zudem über eine Buchse nach außen geführt. Ein einfacher „Batteriedummy“, der sich in ein Standardladegrät platzieren lässt und mittels eines Kabels und eines Steckers die Ladekontakte des Ladegeräts mit der Buchse des *Pacemates* verbindet, erlaubt, dass ein *Pacemate* gemäß der Anforderung ohne Öffnen des Gehäuses einfach aufgeladen werden kann.

Wie beschrieben muss es möglich sein, Daten aus dem Sensornetz auslesen und in dieses einspeisen (beispielsweise zum Programmieren) zu können. Dazu sind 14 *Pacemates* zusätzlich mit einer seriellen Schnittstelle (RS232) versehen. Zwei Datenleitungen (eine zum Senden und eine zum Empfangen) sind über ein Kabel nach außen geführt. Mittels eines Pegelwandlers wird das Signal von den ±3 V auf ±12 V gehoben. Der Pegelwandler wird dabei über einen USB-Anschluss mit dem zusätzlich benötigten Strom versorgt. Die Kommunikation über die serielle Schnittstelle erlaubt dabei Datenraten von 115,2 kbit/s

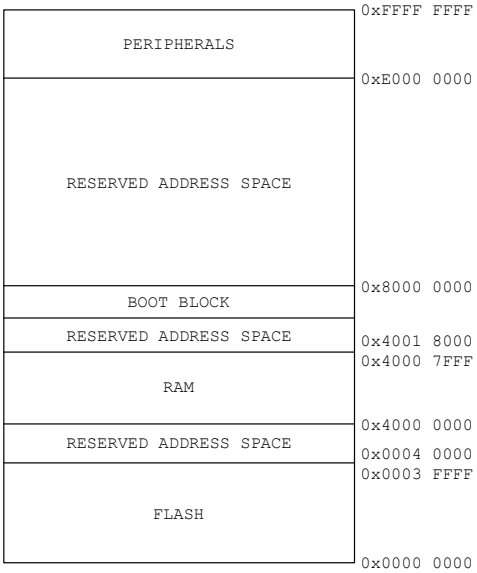


Abbildung 3.5: Speicherplan des Philips LPC2136 Mikrocontroller

und ist dabei so groß wie die Datenrate über das Funkmedium. Zusätzlich liegt bei der *Pacemate*-Plattform eine weitere Kommunikationsleitung auf einem dritten Pol in der Ladebuchse. Dies ermöglicht eine direkte Kommunikation über RS232 mit einem beliebigen *Pacemate*. Da in der Ladebuchse jedoch nur ein Pol zur Verfügung steht wird manuell zwischen Senden und Empfangen umgeschaltet. Mit dem gleichen Pegelwandler ergänzt um einen Umschalter und mit passendem Stecker für die *Pacemate*-Buchse kann über diese Leitung unidirektional mit einem *Pacemate* kommuniziert werden. Ob dabei aus Sicht des *Pacemates* gesendet oder empfangen wird, hängt dabei von der Einstellung des Schalters an dem Wandler ab. Diese Schnittstelle der *Pacemate*-Plattform wird von dem in Kapitel 4 vorgestellten service-orientierten Betriebssystem ausgenutzt.

3.4 Firmware

In dem *Pacemate* befinden sich wie oben beschrieben verschiedene Komponenten wie beispielsweise Mikrocontroller, Funkschnittstelle, Schnittstelle zu einem Pulsgurt, Display und Akkumulator. Alle diese Komponenten bieten Funktionalitäten an, die von Algorithmen, Protokollen und Anwendungen genutzt werden sollen. So soll es möglich sein, beispielsweise den aktuellen Sensorwert (Pulswert) auszulesen genauso wie den aktuellen Ladestand des Akkumulators. Eine sogenannte Firmware bietet die Schnittstellen zu den Grundfunktionalitäten der Hardwarekomponenten. Im Sinne einer Schichtenarchitektur stellt die Firmware die sogenannte Hardwareabstraktionsschicht (engl.: *Hardware Abstraction Layer*, HAL) dar. Im Folgenden werden der Aufbau und die Schnittstellen der *Pacemate*-Firmware beschrieben.

3.4.1 Aufbau

Die *Pacemate*-Firmware besteht aus einer Sammlung von C-Modulen. Dabei wird jede Hardwarekomponente durch ein C-Modul repräsentiert. Die Firmware ist dabei aufgebaut wie in Abbildung 3.6 dargestellt. Der Einstiegspunkt in jedes *Pacemate*-Programm liegt dabei in dem Modul `main.c` in der Funktion `int main()`. In dieser Funktion werden zunächst alle Komponenten initialisiert indem die entsprechenden `Boot()`-Funktionen der Module aufgerufen werden, wie in Auflistung 3.1 aufgeführt. Nach dieser Initialisierung wird die eigentliche Applikation[4], die über der Hardwareabstraktionsschicht liegt, aufgerufen.

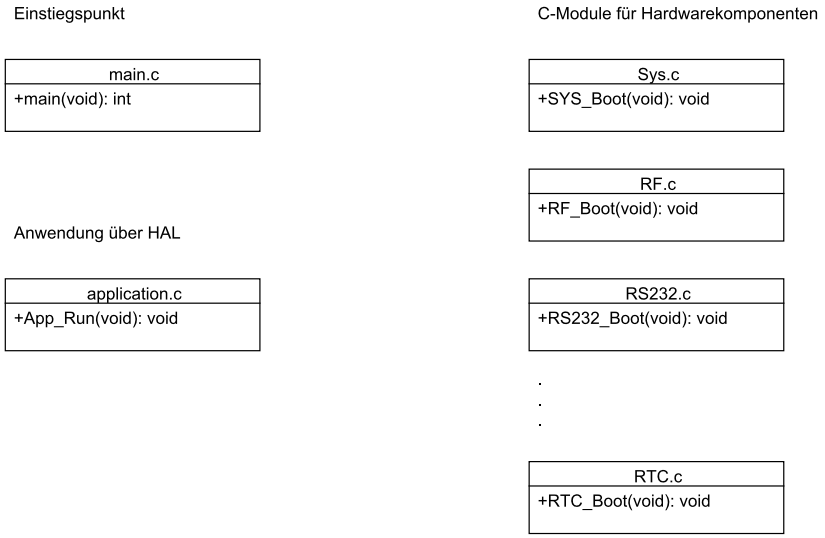


Abbildung 3.6: Aufbau der *Pacemate*-Firmware mit C-Modulen für einzelne Komponenten

Die Applikation selbst ist ereignisgesteuert. Dies bedeutet, dass verschiedene Ereignisse von der Firmware registriert werden und die Applikation daraufhin reagiert. Die aufgerufene Applikation besitzt eine nichtterminierende Hauptschleife, die in Auflistung 3.2 aufgeführt ist. In dieser Schleife werden insbesondere die beiden Systemmethoden `SYS_Loop()` und `SYS_TriggerWatchDog()` aufgerufen. In der Funktion `SYS_Loop()` wird geprüft, ob an den einzelnen Hardwarekomponenten Ereignisse vorliegen. Ist dies der Fall, wird eine zuvor von der Applikation bei der Firmware registrierte Rückruffunktion (engl.: *callback function*) aufgerufen, der eine numerische Repräsentation des

[4]Der Begriff Applikation bezieht sich hierbei auf die Schicht oberhalb der Hardwareabstraktionsschicht und kann dabei wie in Kapitel 4 auch ein Betriebssystem darstellen.

```
1  int main(void)
2  {
3      SYS_Boot();
4      RF_Boot();
5      RS232_Boot();
6      [...]


18     RTC_Boot();
19     App_Run(); //and never return
20     return 0;
21 }
```

Quelltext 3.1: Einstiegspunkt in die *Pacemate*-Firmware mit Aufruf der Applikation

Ereignisses übergeben wird. Hier werden basierend auf dem aktuellen Status der Applikation das Ereignis ausgewertet und gegebenenfalls weitere Funktionen gerufen. Der kontinuierliche Aufruf der Funktion `SYS_TriggerWatchDog()` bewirkt, dass ein sogenannter Watchdog-Timer immer wieder zurückgesetzt wird. Erfolgt in einem festgelegten Zeitraum kein Aufruf dieser Funktion und der Watchdog-Timer läuft ab, so geht die Firmware davon aus, dass sich das Programm in einem Fehlerzustand befindet (beispielsweise in einer Endlosschleife). Die Firmware stellt den *Pacemate* daraufhin aus. So ist gewährleistet, dass die Geräte neu gestartet bzw. auch neu programmiert werden können, wenn fehlerhafte Programme auf den Geräten ausgeführt worden sind.

3.4.2 Schnittstellen

Wie bereits im Kapitel 3.3 beschrieben verfügt der *Pacemate* über verschiedene Kommunikationschnittstellen, die einen Datenaustausch mit den Geräten ermöglichen: Funk-

```
1  while(1)
2  {
3      SYS_Loop();
4      SYS_TriggerWatchDog();
5      [...]

7  }
```

Quelltext 3.2: Hauptprogrammschleife einer Anwendung oberhalb der *Pacemate*-Firmware

schnittstelle, serielle Schnittstelle und I2C-Schnittstelle. Des Weiteren gibt es die Benutzerschnittstellen Display, Tasten und eine Leichtdiode (engl.: *Light Emitting Diode* (LED)). Diese Schnittstellen werden jeweils durch einzelne Module von der Firmware gekapselt. Über einzelne Funktionen in den Modulen lassen sich diese Komponenten steuern. Im Folgenden wird auf diese Realisierung der Schnittstellen in der Firmware und ihre Funktionsweise eingegangen.

Kommunikationsschnittstellen

Die wichtigste Schnittstelle eines jeden Sensorknotens ist die Funkschnittstelle. Das Funkmodul vom Typ DP1205 der Firma Semtech des *Pacemates* ist über einen seriellen Bus, die sogenannte SPI-Schnittstelle (*Serial Peripheral Interface*), mit dem Mikrocontroller verbunden. Die Funktionalität ist dabei von der Firmware in dem Modul `RF.c` gekapselt. Dieses Modul bietet dabei unter anderem Funktionen zum Senden von Daten (`RF_Send(...)`) und zum Empfangen von Daten (`RF_Receive(...)`).

Die Signatur der Senderoutine `RF_Send(...)` dargestellt in der Auflistung 3.3 sieht dabei vor, dass die Funktion die Zieladresse, die Länge der Daten sowie die Speicheradresse, an der die Daten liegen, übergeben bekommt. Die Zieladresse wird dabei durch eine zwei Byte große Zahl angegeben, die der Seriennummer des Empfänger *Pacemates* entspricht. Durch die Angabe der Adresse `0xFFFF` kann ein Paket an alle empfangenden Knoten adressiert werden. Die Funktion `RF_Send(...)` übernimmt dabei keine Wegewahlaufgaben. Es wird lediglich das Paket auf das Funkmedium gegeben.

```
499 pm_boolean RF_Send(pm_uint16_t destaddr, pm_uint8_t length,
        pm_uint8_t* data)
500 {
501     [...]

665 }
```

Quelltext 3.3: Signatur der Funktion `RF_Send(...)` in dem Modul `RF.c`

Die Firmware verwendet beim Versenden dabei einen Protokollrahmen, der eine Präambel zur Synchronisation zwischen Sender und Empfänger, die Adresse des Senders, die Adresse des Empfängers, die Datenlänge sowie eine 16-bit-CRC-Prüfsumme (*Cyclic Redundancy Check*) enthält. Nachdem die Daten versendet wurden, wird ein Ereignis ausgelöst. Dabei wird zwischen einem erfolgreichen Versenden (`EV_RF_TX`) und einem fehlerhaften Versenden (`EV_RF_TX_FAILED`) unterschieden.

Empfängt ein *Pacemate* eine solche Nachricht über Funk, so wird auf diesem Empfänger *Pacemate* ein erneutes Ereignis ausgelöst (`EV_RF_RX`). Dabei ist weder überprüft, ob es sich dabei um ein Paket handelt, welches an dieses Gerät addressiert ist, noch ob die Nachricht korrekt empfangen wurde, d.h. ob die mitgesendete CRC mit der der empfangenen Daten übereinstimmt. Wenn dieses Empfangsereignis von der Firmware registriert

und an die Applikation weitergegeben wird, kann seitens der Applikation das Paket mittels der Firmware über die Methode `RF_Receive(...)` (Auflistung 3.4) abgeholt werden.

```
667 void RF_Receive(pm_uint16_t* srcaddr, pm_uint16_t* destaddr,
        pm_uint8_t* length, pm_uint8_t* data, pm_boolean* crc)
668 {
669   [...]

762 }
```

Quelltext 3.4: Signatur der Funktion `RF_Reveive(...)` in dem Modul `RF.c`

Die Applikation erhält neben den Daten somit die Adresse des Absenders, die des vorgesehenen Empfängers sowie ein Flag, ob die CRC-Prüfung ein korrekt empfangenes Paket signalisiert oder auf Fehler in den Daten hinweist. Es obliegt nun der Applikation, das Paket weiterzuverarbeiten oder zu ignorieren. Dieser Mechanismus, das Paket auf jeden Fall – auch bei anderer Adressierung oder Übertragungsfehler – an die über der Firmware liegende Applikation zu geben, ermöglicht Statistiken zu führen, die für die Interpretation der Anwendung von Nutzen sind. So kann beispielsweise das Nachrichtenaufkommen registriert werden, es können die Übertragungsfehler gemessen werden und es können Statistiken über Nachbarschaftsgrößen erhoben werden.

Wie in Kapitel 3.3 beschrieben, ist neben den mit einem speziellen seriellen Kabel ausgestattenen *Pacemates* bei allen *Pacemates* die Möglichkeit zur unidirektionalen seriellen Kommunikation über den zusätzlichen Kontakt in der Ladebuchse gegeben. Diese Schnittstelle wurde ursprünglich ausschließlich zur Fehlersuche (engl.: *Debugging*) vorgesehen. Daher ist die Schnittstelle im Modul `Debug.c` gekapselt. Dieses Modul bietet Funktionen zum Schreiben (`Debug_RS232_Write(...)`) und Lesen (`Debug_RS232_Read(...)`) auf der seriellen Schnittstelle. Beim Schreiben des letzten Bytes sowie beim Empfangen mindestens eines Bytes werden von der Firmware Ereignisse ausgelöst (`EV_UART1_TX`, `EV_UART1_RX`). Da es sich bei der seriellen Kommunikation um eine strombasierte Kommunikation handelt (im Gegensatz zu einer nachrichtenbasierten Kommunikation wie bei der Funkschnittstelle), realisiert das Modul einen Ringpuffer, in dem die Daten bis zum Versenden vorgehalten werden. Das Modul realisiert daher auch keinen Protokollrahmen. Eine nachrichtenbasierte Kommunikation muss seitens der Applikation oberhalb des `Debug.c`-Moduls realisiert werden.

Der *Pacemate* verfügt weiterhin über eine I^2C-Schnittstelle (engl.: *Inter-Integrated Circuit*). Diese Schnittstelle ist für Erweiterungen des *Pacemates* auf leicht zugängliche Lötstellen auf die Platine geführt, ist jedoch im Rahmen dieser Arbeit nicht verwendet worden. Die I^2C-Schnittstelle wird seitens der Firmware über ein entsprechendes Modul `I2C.h` integriert, jedoch existiert keine Implementierung dieser Schnittstelle.

Benutzerschnittstelle

Eine Eigenschaft, welche die *Pacemate*-Sensorknotenplattform von anderen Plattformen, wie beispielsweise den in Kapitel 3.1 genannten, unterscheidet, ist die Tatsache, dass der *Pacemate* über eine großzügige Anzeige sowie Tasten verfügt. Zudem verfügt der *Pacemate* über eine Leuchtdiode. Mittels dieser Benutzerschnittstellen, ist es zum einen möglich, vielseitige Anwendungen zu schreiben, die einem Benutzer attraktive Dienste anbieten können (siehe MarathonNet-Szenario). zum anderen ermöglichen diese Schnittstellen einen Blick in den aktuellen Systemzustand des einzelnen Sensorknoten, was das Verständnis der Dynamik von verteilten Anwendungen erleichtern und verbessern soll. Mittels der Tasten ist zudem eine genaue Steuerung und Kontrolle einer Anwendung möglich, was sowohl für den Benutzer attraktiv, aber auch für die Steuerung von Experimenten unabdingbar ist.

Genau wie die Kommunikationsschnittstellen werden die Benutzerschnittstellen durch einzelne C-Module von der Firmware gekapselt. Die Anzeige wird dabei durch das Module `Display.c` gesteuert. Dieses Modul stellt Funktionen zur Anzeige von Text (`Display_Text(...)`) und einfachen grafischen Elementen wie Punkten und Rechtecken (`Display_Pixel(...), Display_Rectangle(...)`) bereit. Die Informationen über die „neue“ Bildschirmanzeige werden zunächst in einen Speicherbereich geschrieben. Dieses Abbild der darzustellenden Pixel wird durch den Aufruf der Funktion zur Aktualisierung des Bildschirms (`Display_Update(...)`) auf die Anzeige übertragen.

Gerade durch die Ausgabe von Text, können zu Test- und Forschungszwecken aktuelle Werte von Variablen und freiem Speicher auf dem Bildschirm ausgegeben werden. Somit kann die Funktion insbesondere von verteilten Algorithmen leichter getestet werden. Um jedoch das Verhalten mehrerer Geräte schnell beurteilen zu können, muss das Gerät eine visuelle Rückmeldung geben können, die leicht auch aus Entfernung zu erkennen ist. Hierzu dient die Leuchtdiode des *Pacemates*. Das C-Module `Led.h` bietet dazu die Funktionen die Diode einzuschalten (`Led_On(...)`), auszuschalten (`Led_Off(...)`) und umzuschalten (`Led_Toggle(...)`).

Die Möglichkeit mit dem *Pacemate* über Tasten zu interagieren erlaubt ein hohes Maß an Flexibilität bei der Gestaltung von Anwendungen und bei der Steuerung von Experimenten. So kann man beispielsweise Knoten während eines Experimentes in einen bestimmten Modus schalten, um Experimente mit verschiedenen Parametern durchführen zu können. Das Bedienen der Tasten erzeugt dabei einzelne Ereignisse. So erzeugt das Drücken einer Taste ein eigenes Ereignis (`EV_KEYx_PRESSED`, wobei `x` der Index der gedrückten Taste ist) und auch das Loslassen der Taste (`EV_KEXx_RELEASED`). Dies ermöglicht das Erfassen von Tastendrücken verschiedener Länge und Frequenz sowie gleichzeitige Tastendrücke. Diese Erfassung muss jedoch seitens der über der Firmware liegenden Applikation realisiert werden.

3.5 Ergebnis

In diesem Kapitel wurden zunächst verschiedene existierende Sensorknotenplattformen vorgestellt. Dabei wurde deutlich, dass sich die Plattformen in Funkmodulen und zur Verfügung stehendem Speicher unterscheiden. Vor dem Hintergrund des MarathonNet-Anwendungsszenarios, bei dem Athleten bei einer Laufveranstaltung mit Sensorknoten ausgestattet werden sollen, wurden verschiedene anwendungsspezifische Anforderungen sowie forschungsspezifische Anforderungen formuliert.

Auf dieser Grundlage hat der Autor die *Pacemate*-Sensorknotenplattform entwickelt, welche für praktische Einsätze unter Nicht-Laborbedingungen geeignet ist. Im Vergleich zu den existierenden Plattformen bietet die *Pacemate*-Plattform ein robustes Gehäuse, welches zudem ergonomischen Richtlinien genügt. Weiterhin verfügt diese Plattform über ein Display sowie leicht erkennbare Tasten. Zudem verfügt der *Pacemate* im Vergleich zu anderen Sensorknoten über einen großen Speicher sowohl für Programme im Flash, wie auch RAM.

Aus den genannten Eigenschaften ergeben sich Vorteile für Anwendung und Forschung. Insbesondere das robuste Gehäuse ermöglicht eine einfache Handhabung der Sensorknoten in der Anwendung. Die *Pacemate*-Sensorknoten können an Testpersonen verteilt werden, ohne dass diese spezielle Rücksicht auf die Technik nehmen müssen, wie es bei einem als bloße Platine vorliegenden Sensorknoten der Fall wäre. Feuchtigkeit sowie physikalische Einwirkungen werden durch das Gehäuse von der Technik ferngehalten. Es können viele Personen mit einem *Pacemate* ausgestattet werden, ohne dass es eine spezielle Einweisung in die Handhabung der Geräte geben muss.

Die Tatsache, dass die Geräte einfach an viele Personen verteilt werden können, ist die Grundlage für die Evaluation von mobilen Szenarien. Durch die große Anzeige und die Tasten lassen sich vielseitige Applikationen erstellen, die den Trägern der Geräte verschiedene Dienste anbieten können. So lassen sich leicht viele freiwillige Teilnehmer finden, die zum einen an aktueller Forschung teilnehmen, jedoch gleichzeitig auch persönlich durch die gebotenen Dienste profitieren. Auf diese Weise lassen sich Sensorknoten in der Größenordnung von bis zu 500 Stück für eine reales Testszenario bewegen und es können praktische Forschungsergebnisse erzielt werden.

Die Anzeige bietet den weiteren Vorteil, dass sich Algorithmen und Protokolle auf den Geräten testen lassen. Mittels des Displays kann der aktuelle Status verschiedener Algorithmen angezeigt werden. Insbesondere kann man somit Einblick in die Dynamik des Sensornetzes erhalten, welche nicht zwingend eins zu eins in einen Simulator abgebildet werden kann. Insbesondere kann man mittels der Anzeige leicht den Status jedes einzelnen Knotens überprüfen, ohne diesen zusätzlich über eine serielle Schnittstelle mit einem Rechner zu verbinden. Weiterhin erlaubt die Anzeige eine hardwarenahe Entwicklung, die beispielsweise das Wissen über die aktuelle Belegung einzelner Speicherbereiche benötigt. Die Bereiche von Interesse können auf der Anzeige ausgegeben werden. Diese Eigenschaft war insbesondere bei der Erstellung des im nächsten Kapitel vorgestellten service-orientierten Betriebssystems von großem Nutzen. Parallel zu den hier präsen-

tierten Forschungsarbeiten kam die *Pacemate*-Plattform auch in anderen Arbeiten zum Einsatz. So wurde beispielsweise ein funkbasiertes Umfragesystem [26] mit der *Pacemate*-Plattform realisiert. Außerdem wurde die *Pacemate*-Plattform zur Bewegungsklassifikation in medizinischen Anwendungen [33] verwendet.

4 Surfer OS – Service-orientiertes Betriebssystem für Sensorknoten

Im vorigen Kapitel wurde die Sensorknotenplattform *Pacemate* vorgestellt. Auf dieser Plattform bietet eine Firmware die Softwareschnittstellen in Form von C-Modulen für die einzelnen Hardwarekomponenten. Diese Firmware bildet jedoch lediglich eine Hardwareabstraktionsschicht und bietet somit nur grundlegende Dienste, die für die Erstellung komplexerer Anwendungen nicht ausreichen. Beispielsweise realisiert die Firmware keine Wegewahl in dem durch die Sensorknoten entstehenden Netz. Die Firmware der *Pacemates* realisiert für die Kommunikation lediglich die unteren beiden Schichten des in Kapitel 2.1 vorgestellten ISO-OSI-Modells, die Physikalische Schicht und die Sicherungsschicht. So kann die Anwendung auf Knoten A nicht mit der Anwendung auf Knoten B sprechen, wenn diese nicht direkte (1-hop) Nachbarn sind. Eine Multi-hop-Kommunikation ist also bisher nicht möglich und steht beispielhaft für weitere Dienste, die für die Entwicklung von Anwendungen notwendig sind.

Dem Anwendungsprogrammierer sollen diese Mechanismen jedoch verborgen bleiben, da diese ein genaues Verständnis von verteilten Systemen sowie Sensornetzen voraussetzen. Daher wird eine Schicht zwischen Anwendung und Hardwareabstraktionsschicht benötigt, die neben der Wegewahl dem Entwickler auch weitere Dienste bietet. Ein Betriebssystem (engl.: *Operating System* (OS)) oder eine Zwischenanwendung (engl.: *Middleware*) für Sensorknoten wird benötigt, welches diese Aufgabe erfüllt und dem Anwendungsentwickler nützliche und komplexe Dienste anbietet.

Gerade bei dem Problem der Wegewahl wird jedoch deutlich, dass das zu verwendende Wegewahlverfahren stark von der Topologie des Netzes, dem zu erwartenden Datenaufkommen und somit von dem Anwendungsszenario bzw. der Anwendung selber abhängt. Verschiedene Verfahren eignen sich besser für Netzwerke mit hoher Dichte, andere würden dort eine zu große Anzahl unnötiger Nachrichten erzeugen, was zu einer Auslastung des Funkmediums führen würde. Wiederum andere Verfahren eignen sich nicht für mobile Szenarien, in denen die Nachbarschaften ständigen Veränderungen unterliegen.

Weiterhin spielt der Aspekt der knappen Ressourcen auf Sensorknoten eine wichtige Rolle. Dienste, die von der Anwendung nicht benutzt werden, benötigen Speicherplatz, welcher wiederum von der Anwendung zur Auslagerung von erhobenen Daten genutzt werden könnte. Aus diesen Gründen sind die Komponenten, die ein Betriebssystems bereitstellen sollte, direkt abhängig von der darüberliegenden Applikation. Das Betriebssystem sollte somit für die Anforderungen einer speziellen Anwendung optimiert werden.

Wie bereits in Kapitel 1.1 erläutert können sich jedoch die Anforderungen an eine Anwendung sowie die zugrundeliegende Netztopologie während des Betriebs des Sensornetzes verändern. Dadurch verändern sich auch die Anforderungen an das Betriebssystem der Sensorknoten.

Weiterhin muss es möglich sein, dem Anwendungsprogrammierer Entscheidungen abzunehmen, wie beispielsweise welches Wegewahlprotokoll verwendet wird. Diese Entscheidungen bedürfen wiederum der Fachkenntnis, die bei dem Anwendungsentwickler nicht vorausgesetzt werden soll. Zudem hat sich in den in Kapitel 2.1.3 vorgestellten Arbeiten gezeigt, dass die Vermessung von Funkcharakteristiken sehr aufwendig ist. Diese bilden jedoch mit die Basis, auf der die Entscheidungen über die zu verwendenden Algorithmen und Protokolle getroffen wird. Demgegenüber kann das Sensornetz selbstständig aktuelle Netzparameter wie beispielsweise durchschnittliche Nachbarschaftsgrößen und Topologieänderungen bestimmen. Vor diesem Hintergrund ist das Sensorknotenbetriebssystem *Surfer OS* [61] im Rahmen dieser Arbeit entwickelt worden.

In diesem Kapitel werden zunächst existierende Betriebssysteme und Middlewares vorgestellt. Dabei werden die Begriffe Middleware und Betriebssystem zuvor voneinander abgegrenzt. Im Anschluss wird das *Surfer OS* vorgestellt. Es werden die Anforderungen identifiziert und das Grundkonzept des Betriebssystems vorgestellt. Basierend darauf werden die Architektur und die Systemkomponenten beschrieben. Darauffolgend wird die Implementierung des *Surfer OS* vorgestellt. Dabei wird insbesondere auf die Serviceverwaltung (engl.: *Service Management*), die Speicherverwaltung (engl.: *Memory Management*) und die Aufgaben- bzw. Prozessverwaltung (engl.: *Task Management*) als Hauptbestandteile eingegangen. Das Kapitel schließt mit einer Zusammenfassung und Bewertung der Ergebnisse.

4.1 Verwandte Arbeiten

Im Folgenden werden im Rahmen eines Überblickes verschiedene Arbeiten vorgestellt, die für den Anwendungsprogrammierer Dienste auf dem Sensorknoten realisieren und zugänglich machen. Dabei wird in manchen Arbeiten von Betriebssystemen gesprochen, in anderen von Middleware. Dies wird zum Anlass genommen, zunächst beide Begriffe von einander abzugrenzen.

Tanenbaum [100] definiert zwei Aufgaben für ein Betriebssystem: zum einen sieht er das Betriebssystem als erweiterte Maschine, zum anderen als Betriebsmittelverwalter. Die erweiterte Maschine präsentiert dem Benutzer eine Maschine, die für ihn leichter zu programmieren ist als die darunterliegende Hardware. Technische Details der Hardware werden dadurch vor dem Benutzer verborgen. Das Betriebssystem als Betriebsmittelverwalter koordiniert und kontrolliert die Zuteilung der auf dem Gerät zur Verfügung stehenden Betriebsmittel (engl.: *ressourcen*) und deren Zusammenspiel. Es garantiert hierdurch insbesondere den fehlerfreien Ablauf nebenläufiger Prozesse.

Eine Middleware wird von Tanenbaum in [102] als eine Schicht oberhalb eines Betriebssystems definiert, deren Aufgabe darin besteht, die Verteilungstransparenz zu erhöhen und somit die Verteiltheit des Systems vor dem Benutzer zu verbergen.

Im Falle der Sensorknoten gibt es die Situation, dass ein Sensorknoten allein keine Aufgaben erfüllen kann. Es ist inhärent für ein Sensornetz, dass etliche Sensorknoten interagieren. Somit können die Aufgaben der Middleware direkt in das Betriebssystem für einen Sensorknoten übernommen werden. Daher werden im Folgenden sowohl Betriebssysteme wie auch Middlewares gleichermaßen betrachtet. Zunächst wird auf die Sensorknotenbetriebssysteme TinyOS, Contiki, Mantis und SOS eingegangen. Danach wird ein Überblick über die verschiedenen Middlewareansätze Maté, Cougar, TinyDB, DSWare und Hood gegeben.

4.1.1 TinyOS

Das Sensorknotenbetriebssystem TinyOS [35][37][56][57] wurde zusammen mit der in Kapitel 3.1.1 vorgestellten Sensorknotenplattform MICA an der University of California in Berkeley entwickelt. Man entschied sich, ein neues Betriebssystem zu entwickeln, da bestehende Betriebssysteme für eingebettete Systeme den speziellen Rahmenbedingungen auf den Sensorknoten, wie geringer Speicherplatz, begrenzte Energieressourcen und hohe Nebenläufigkeit nicht genügten [35]. TinyOS ist in der dritten Generation und wurde bereits 2005 von über hundert Forschungsgruppen und in industriellen Produkten verwendet [57]. TinyOS ist frei verfügbar unter der offiziellen Internetpräsenz [110]. Aktuell ist TinyOS ein sogenanntes *open source* Betriebssystem für Sensornetze an dessen Weiterentwicklung und Portierung jeder mitwirken kann.

TinyOS setzt sich aus etlichen verschiedenen Modulen zusammen. Der feingranulare Ansatz soll dabei dem Problem der geringen Speicherressourcen auf den Sensorknoten begegnen. TinyOS wird zusammen mit der Applikation für die Zielplattform übersetzt. Im resultierenden Programm werden lediglich solche Module des Betriebssystems verwendet, die auch von der Applikation genutzt werden. Man erhält ein applikationsspezifisches Betriebssystem welches speziell auf die Anforderungen der Anwendung optimiert ist und somit Speicherplatz für nicht verwendete Module einspart. TinyOS wird daher von seinen Entwicklern nicht nur als ein Betriebssystem sondern als ein *programming framework* [57] gesehen.

Der Aspekt der Nebenläufigkeit ergibt sich aus Anwendungsanforderungen, wie das simultane Auslesen von Daten verschiedener Sensoren bei gleichzeitigem Versenden durch das Netz und Weiterleiten bzw. Aggregieren von Daten anderer Knoten. TinyOS realisiert dabei ein ereignisbasiertes Programmiermodell, welches einen hohen Grad an Nebenläufigkeit erlaubt. So können sogenannte lang laufende Hintergrundaufgaben (engl.: *background tasks*) durch Hardware-Ereignisse unterbrochen werden, die sofort abgearbeitet werden.[1] Der ereignisbasierte Ansatz soll dabei die knappen Energieressourcen auf den Sensorknoten schonen. Ein immer wiederkehrendes Abfragen von Ereignissen (engl.:

[1] Dies ist in der *Pacemate* Firmware wie in Kapitel 3.4 ebenfalls durch Ereignisse realisiert.

polling) ist dabei nicht erlaubt. So kann der Mikrocontroller in ungenutzten CPU-Zyklen in einem sogenannten Schlafstatus verbringen, in dem die Stromaufnahme geringer ist als im aktiven Betrieb. Erst durch ein auftretendes Hardware-Ereignis wird der Mikrokontroller wieder in den Betriebsstatus gebracht.

Sowohl TinyOS selbst wie auch die Applikationen sind in einer speziell für eingebettete vernetzte Systeme entwickelte Programmiersprache *nesC* [25] geschrieben. Diese ermöglicht dabei eine Reduktion des resultierenden Programmcodes und liefert spezielle Abstraktionen für das ereignisbasierte Programmieren, was die Entwicklung von Applikationen vereinfachen soll.

4.1.2 Contiki

Das Betriebssystem Contiki [18] wurde am Swedish Institute of Computer Science entwickelt und 2004 von Dunkels et al. vorgestellt. Contiki ist wie TinyOS ebenfalls ein ereignisbasiertes Betriebssystem, welches speziell für die Rahmenbedingungen auf eingebetteten vernetzten Systemen entwickelt worden ist. Contiki erlaubt jedoch im Vergleich zu TinyOS das dynamische Laden von Programmen und Diensten [17]. Contiki war zunächst nur für die Sensorknotenplattform Scatterweb ESB (engl.: *Embedded Sensor Board*) der Firma Scatterweb GmbH [92] in Zusammenarbeit mit der Freien Universität Berlin verfügbar. Diese Plattform verwendet den Mikrocontroller MSP430 der Firma Texas Instruments [104], welcher ebenfalls wie der im *Pacemate* verwendete Philips LPC2136 Bereiche des Flash-Speichers während der Ausführung eines Programms reprogrammieren kann. Aktuell werden auch weitere Plattformen von Contiki unterstützt. Contiki ist ebenfalls wie TinyOS ein *open source* Projekt und ist auf der eigenen Internetpräsenz [16] frei verfügbar. Das Betriebssystem kommt bereits in verschiedenen Anwendungen zum Einsatz, beispielsweise zur Tunnelüberwachung [75], Einbrucherkennung [16] und Hochseeüberwachung [114].

Der Betriebssystemkern von Contiki besteht aus einen Steuerprogramm (engl.: *scheduler* – dieser Begriff wird im Folgenden synonym verwendet), der die Ereignisse an laufende Prozesse weitergibt. Zusätzlich implementiert Contiki im Gegensatz zu TinyOS einen Abfragemechanismums (engl.: *polling mechanism*). Dieser ermöglicht eine Priorisierung von geplanten Ereignissen. Ein Wechseln des Mikrocontrollers in den Schlafstatus ist daher vom Betriebssystem nicht vorgesehen kann jedoch seitens der Applikation realisiert werden. Das Steuerprogramm gibt dazu Auskunft über die geplanten anstehenden Ereignisse. Wenn der Prozessor durch ein Ereignis (engl.: *interrupt*) wieder in den Betriebsmodus wechselt, wird der Abfragemechanismus wieder gestartet.

Contiki adressiert die Notwendigkeit dynamischen Nachladens von Programmen und greift auch die Idee der Services auf. Hierbei steht insbesondere das Beheben von Fehlern in der Software beim Einsatz von großen Sensornetzen im Vordergrund, wo das manuelle Einsammeln und neu Programmieren aller Knoten nicht möglich ist. Dunkels et al. definieren einen Service dabei als einen Prozess, der eine Funktionalität implementiert, die von anderen Prozessen genutzt werden kann. Dabei nutzt die Anwendung eine soge-

nannte *stub library*, um mit einen Service zu kommunizieren. Die *stub library* nutzt einen *service layer*, der den Service auffindet. Hierbei werden Versionsnummern zwischen der *stub library* und dem Service abgeglichen, um die Kompatibilität der zwischen Applikation und der vorhandenen Serviceversion sicherzustellen. Contiki erlaubt die Übergabe von Adresszeigern zwischen dem neuen und dem zu ersetzenden Service, was ermöglichen soll, einen Service auf einem Knoten zu erneuern, ohne den lokalen Status des Services zu verlieren. Dies ist insbesondere beim Ersetzen von fehlerhaften Diensten durch aktuellere Versionen von Vorteil. Das neu Platzieren von Programmcode (engl.: *relocation*) bzw. das dynamische Auflösen von Adressen (engl.: *dynamic linking*) innerhalb der nachgeladenen Programme und Services wird dabei über ein spezielles kompaktes ELF (engl.: *Executable and Linking Format*) realisiert.[2] Sowohl Contiki wie auch die einzelnen Programme und Services sind in der Programmiersprache C geschrieben.

4.1.3 MANTIS

Das MANTIS Betriebssystem (*M*ultimodAl *Ne**T*works of *I*n-situ *S*ensors) [1] wurde von der University of Boulder in Colorado im Jahr 2003 veröffentlicht. MANTIS wurde mit dem Ziel entwickelt, den Entwicklern eine einfachere Abstraktion zum schnellen Erstellen einfacher Sensornetzanwendungen zu bieten. Außerdem wollte man dem Entwickler die Möglichkeit geben, Knoten drahtlos zu reprogrammieren und auf entfernten Knoten Fehler suchen und beseitigen zu können. Dabei soll MANTIS gleichzeitig den Ressourcenbeschränkungen auf den Sensorknoten genügen. MANTIS wurde zunächst auf einer eigens entwickelten Plattform realisiert, dem *nymph* Sensorknoten, der wie die in Kapitel 3.1.1 vorgestellten MICA Sensorknoten den Mikrocontroller Atmel Atmega 128 verwendet. Aktuell existieren weitere Portierungen unter anderem auch für die verschiedenen MICA Plattformen, die auf der Internetpräsenz des MANTIS Projekt [73] verfügbar sind. Das MANTIS Betriebssystem wurde unter anderem in einem Projekt zur Überwachung von Wetterbedingungen und Waldbränden [30] im Bitterroot National Forest in Idaho, USA im Jahr 2005 verwendet

Das MANTIS Betriebssystem besteht aus dem Betriebssystemkern sowie einem Scheduler. MANTIS erlaubt dabei, dass verschiedene leichtgewichtige Prozesse (engl.: *threads*) gleichzeitig aktiv sind. Der Scheduler erlaubt dabei für die verschiedenen Threads Prioritätstufen und verteilt die Ressourcen in einem Rundlaufverfahren (engl.: *round-robin*). Über dem Betriebssystemkern und dem Scheduler ist in MANTIS eine Programmierschnittstelle (engl.: *Application Programming Interface* (API)) realisiert. Die API stellt dabei eine umfangreiche Sammlung von Funktionen zur Ein- und Ausgabe (I/O) sowie zum Zugriff auf Systemressourcen zur Verfügung. MANTIS ist dabei wie Contiki in der Programmiersprache C implementiert. Durch Vergabe von Ressourcen sowie durch das Bereitstellen einer C-Programmierschnittstelle will MANTIS dem Anwendungsprogrammierer eine Abstraktion des Sensorknotens bieten, die einem Arbeitsplatzrechner gleicht. Somit soll dem Programmierer das Einarbeiten und somit das Erstellen von Sensornetzanwendungen vereinfacht werden.

[2] Auf dieses Verfahren zur Codemigration wird in Kapitel 5 im Detail eingegangen.

4.1.4 SOS

Das Betriebssystem SOS [29] wurde 2005 von der University of California in Los Angeles vorgestellt. SOS ist ein dynamisches Betriebssystem speziell für Sensornetze. Es erlaubt Laufzeitanpassungen der Anwendung durch die Nutzung von dynamisch nachladbaren Modulen, die eine bestimmte Aufgabe oder Funktion erfüllen. SOS unterstützt die in Kapitel 3.1.1 vorgestellten MICA2 und MICAZ sowie die Sensorknotenplattform XYZ [69] der Yale School of Engineering and Applied Science. SOS ist verfügbar auf der SOS-Projektseite der University of California [113]. Das Projekt wird jedoch seit November 2008 nicht weitergeführt.

Der Kern des Betriebssystems SOS besteht aus der Hardwareabstraktion, der Einbindungsfunktionalität für dynamische Module, einem Scheduler und einer dynamischen Speicherverwaltung. Die Kommunikation zwischen den Modulen und dem Betriebssystem wird über eine sogenannte Sprungtabelle (engl.: *Jump Table*) realisiert, welche die tatsächliche Adresse der geforderten Funktion referenziert. Das Einbinden und Ausführen der Module wird über positionsunabhängigen Programmcode realisiert. Positionsunabhängiger Code kann an jeder beliebigen Adresse im Speicher platziert und dort ausgeführt werden. Die Unabhängigkeit von der Position im Speicher wird dadurch erreicht, dass Sprünge zu absoluten Speicheradressen durch relative Sprünge zum Befehlszähler (engl.: *program counter*), der auf die aktuell ausgeführte Instruktion zeigt, ersetzt werden. Durch die Verwendung von positionsunabhängigen Code ergeben sich jedoch Begrenzungen. So können Module für einen AVR Mikrocontroller wie dem Atmel Atmega die Größe von 4 kbyte nicht überschreiten, da relative Sprünge nicht über maximal 4 kbyte adressiert werden können. Zusätzlich wird die Compileroption für positionsunabhängigen Code nicht auf allen Plattformen unterstützt.

4.1.5 Middlewareansätze

Ein Ziel dieses Kapitels ist es, den Grundstein für eine einfache Abstraktion des verteilten Systems, nämlich des Sensornetzes, zu legen. Nach der genannten Definition realisiert eine Middleware die Transparenz der Verteiltheit. Wie in Kapitel 2.1 beschrieben, realisiert ein Sensornetz bestehend aus Sensorknoten immer ein verteiltes System. Dies hat zur Folge, dass Middlewarekonzepte direkt in ein Betriebssystem für Sensorknoten übernommen werden können. Im Folgenden wird das Konzept der virtuellen Maschine als Middleware für Sensornetze sowie Ansätze, die eine spezielle Abstraktion des Sensonetzes bieten, betrachtet.

Virtuelle Maschinen

Eine virtuelle Maschine erlaubt es dem Programmierer, mittels einer fest definierten API einzelne Module zu schreiben, die dann in das Netzwerk injiziert und auf den Sensorknoten interpretiert werden. Anwendungen können auf diese Weise flexibel gestaltet und somit zur Laufzeit angepasst werden. Zudem bietet eine virtuelle Maschine ein hohes

Maß an Sicherheit gegen Programmierfehler in den Modulen. Die Module werden in einer gegen das System abgeschirmten Laufzeitumgebung ausgeführt (*Sandbox-Prinzip*). Etwaige Fehler können durch die Laufzeitumgebung der virtuellen Maschine abgefangen werden.

Das in 2002 vorgestellte Maté [55] realisiert eine solche virtuelle Maschine für Sensorknoten oberhalb von TinyOS. Maté abstrahiert von dem Betriebssystem und stellt nur ein spezielle Auswahl von Instruktionen zur Verfügung. Ein Modul, das mit diesen Instruktionen geschrieben ist, kann einfach in eine laufende Sensornetzanwendung eingebunden werden. Maté abstrahiert zusätzlich von dem asynchronen Verhalten des Sensornetzes. Bei einer Sendeaktion wird das aktuelle Modul angehalten, bis die Antwort des Kommunikationspartners eingetroffen ist. Somit lassen sich entfernte Funktionsaufrufe (engl.: *Remote Procedure Calls* (RPC)) realisieren. Dies soll dem Programmierer das Erstellen von Applikationen vereinfachen.

Programmierabstraktionen

Weitere Middlewareansätze legen eine bestimmte Art von Anwendung zu Grunde. Auf diese Weise können durch die Middleware spezielle Dienste und Abstraktionen angeboten werden, die für diesen Typ von Anwendungen sinnvoll sind. Insbesondere stellt die Middleware dabei nicht mehr den einzelnen Sensorknoten, sondern das gesamte Sensornetz als eine Entität für den Anwendungsprogrammierer zur Verfügung. Auf diese Weise wird durch die Middleware eine Verteilungstransparenz erreicht.

Die Hauptaufgaben von Sensornetzen sind das Aufzeichnen und Bereitstellen von Sensordaten. So kann ein Sensornetz als eine Art Datenbank angesehen werden. TinyDB [70] und Cougar [125] bieten eine solche Abstraktion. TinyDB ist eine Sensornetzanwendung oberhalb von TinyOS, die von Intel-Research, Berkeley in Zusammenarbeit mit der University of California, Berkeley entwickelt worden ist und auf der Projektpräsenz [112] verfügbar ist. TinyDB bietet dem Anwendungsentwickler die Möglichkeit, Anfragen an das Sensornetz mittels der Sprache TinySQL zu stellen, welche die Anfragesprache für Datenbanken SQL (*Structured Query Language*) um sensornetzspezifische Befehle ergänzt. Dem Programmierer, der nun nicht mehr das Sensornetz selber, sondern lediglich die Auswertungssoftware erstellt, erscheint das Sensornetz wie eine Tabelle einer relationalen Datenbank, in der die Spalten die Attribute der Knoten sind, beispielsweise aktuelle Sensorwerte, Knotennummer etc. Die Verarbeitung der Anfragen sowie die Wegewahl durch das Netz und die etwaige Vorverarbeitung von Daten im Netz wird von TinyDB realisiert und vor dem Programmierer verborgen.

Die Middleware Cougar stellt dem Programmierer das Sensornetz ebenfalls als verteilte Datenbank dar. Die Hauptforschungsaspekte liegen laut der Internetpräsenz des Cougar Projektes [9] weniger auf der Erweiterung der Abstraktion sondern auf der Leistungssteigerung der darunterliegenden Protokolle und Algorithmen. Augenmerk wird besonders auf die effiziente Auswertung von Anfragen sowie die Optimierung des Zusammenspiels zwischen Daten und Wegewahl gelegt.

Neben der Abstraktion, ein Sensornetz für den Benutzer als Datenbank erscheinen zu lassen, gibt es weitere Middlewareansätze, die andere Eigenschaften der Sensornetze in den Vordergrund stellen. Beispielhaft sind hier die Middlewares DSWare [60] und Hood [118] aufgeführt. DSWare stellt das Sensornetz als eine Entität dar, die dem Benutzer Daten liefert, sobald vordefinierte Ereignisse wie beispielsweise ein akustischen Ereignis oder eine Explosion vom Sensornetz registriert werden. Hood realisiert eine Programmierabstraktion basierend auf Nachbarschaftsabstraktionen. Motivation hierfür ist die Tatsache, dass für viele Anwendungen die Nachbarschaften eine große Rolle spielen. Auf diese Weise stellt man dem Programmierer bereits Abstraktionen und Dienste zur Verfügung, die die Erstellung von Anwendungen vereinfachen und ihre Komplexität reduzieren.

4.2 Anforderungen

Die im vorigen Kapitel vorgestellten *Pacemates* verfügen lediglich über eine Firmware, die die Hardwareabstraktion realisiert, welche dem Benutzer eine einfachere Programmierung des einzelnen Knotens ermöglicht. Nach der zuvor aufgeführten Definition von Tanenbaum muss ein Betriebssystem darüber hinaus auch die Betriebsmittelverwaltung kontrollieren. Ein Betriebssystem für Sensorknoten kann dabei zusätzlich die inhärente Verteiltheit des Sensornetzes verbergen. Zudem soll es wie bereits aufgeführt die Flexibilität bieten, dass Anwendungen leicht an sich ändernde Bedingungen und Anforderungen angepasst werden können. Mittels einer so geschaffenen Abstraktion kann somit durch das Betriebssystem die Erstellung von Anwendungen für das gesamte Sensornetz vereinfacht werden. Dadurch kann die Sensornetztechnologie größeren Nutzergruppen zugänglich gemacht werden, die über keine Expertise im Bereich der Sensornetze verfügen. Die auf den *Pacemates* vorhandene Firmware erfüllt diese Anforderungen nicht.

TinyOS bietet zwar die Möglichkeit zum Zusammenstellen des System aus einzelnen Komponenten, jedoch muss der Entwickler vor der eigentlichen Ausbringung des Sensornetzes genau wissen, welche Protokolle für sein Szenario geeignet sind. TinyOS selbst erlaubt keine Anpassung der Anwendung und des Systems, wenn die Sensornetzanwendung im laufenden Betrieb ist. Die TinyOS-Komponenten sind speziell für die gegebene starre Anwendung optimiert. Man hat eine enge Kopplung zwischen Betriebssystemkomponenten und Anwendungskomponenten. Einzig eine Ersetzung der kompletten Applikation, also eine Neuprogrammierung der Knoten mittels spezieller Protokolle wie Deluge [38], ermöglicht eine Anpassung der Applikation. Dabei muss jedoch bei jeder Anpassung die komplette Applikation durch das Netz gesendet werden, was eine hohe Belastung des Funkmediums und einen hohen Energieverbrauch bedeutet. Zudem müssen die Knoten neu gestartet werden, was wiederum den Verlust von Sensordaten und Netzinformationen bedeutet. Wird dies nicht berücksichtigt, so kann dies zu inkonsistenten Netzzuständen führen.

Contiki und SOS erlauben das Nachladen einzelner Module, nutzen dabei aber spezielle Eigenschaften der zugrundeliegenden Architekturen aus, auf die im folgenden Kapitel 5

	TinyOS	Contiki	Mantis	SOS	virtuelle Maschinen	Programmier-abstraktionen
Nachladen von Modulen	–	+	+	+	+	–
Modulintegration ohne Neustart	–	+	–	+	+	–
Anpassbarkeit des Systems	–	–	–	–	–	–
Modulimplementierung	*nesC*	C	C API	C	indiv. API	–
Verteilungstransparenz	–	–	–	∘	∘	+
Abstraktion für den Nutzer	–	–	–	service-orientiert	virtuelle Maschine	bspw. Datenbank
Portierbarkeit	+	∘	+	∘	+	+

(+: vorhanden, –: nicht vorhanden, ∘: teilweise)

Tabelle 4.1: Vergleich der verschiedenen Betriebssysteme und Middlewareansätze für Sensorknoten

im Rahmen der Migrationstechniken eingegangen wird. Unter diesen Voraussetzungen sind Contiki und SOS ebenfalls nicht nutzbar

Mantis erlaubt zwar das Laden einzelner Module zur dynamischen Reprogrammierung, erfordert jedoch den Neustart des Sensorknotens. Zudem liefert Mantis eine Abstraktion, die sich nur auf den einzelnen Knoten und nicht auf das gesamte Sensornetz bezieht. Eine Verteilungstransparenz wird dadurch nicht realisiert.

Die Verwendung eine Middleware realisiert die Verteilungstransparenz, erlaubt jedoch nicht eine Anpassung der unteren Schichten wie beispielsweise des Wegewahlverfahrens auf der Vermittlungsschicht. Dadurch bieten solche Middlewarenansätze nicht die Flexibilität, die benötigt wird, um unter verschiedensten Netzbedingungen, wie beispielsweise Mobilität zu funktionieren. Eine virtuelle Maschine benötigt zudem zusätzliche Rechenzeit zur Interpretation des Programmcodes. Middlewareansätze, die eine spezielle Programmierabstraktion bereitstellen, sind außerdem an ein spezielles Anwendungsszenario gebunden.

Die Eigenschaften der einzelnen Betriebssysteme und Middlewareansätze sind in Tabelle 4.1 gegenübergestellt. Eine Portierung eines der aufgeführten etablierten Betriebssysteme auf die *Pacemate*-Plattform ist nicht zielführend. Die Betriebssysteme bieten weder die gewünschte Flexibilität noch die notwendige Abstraktion, um die Entwicklung und Integration von Applikationen zu vereinfachen. Auf Grund dieser Tatsache wurde die Entscheidung getroffen, mit *Surfer OS* einen neuen Ansatz für ein Sensorknotenbetriebssystem zu realisieren.

Die Entwicklung eines neuen Sensorknotenbetriebssystems ermöglicht es, ein System zu entwickeln, das aus lose gekoppelten Diensten besteht. Insbesondere in Sensornetzen können die Anforderungen an das Betriebssystem in Abhängigkeit vom Anwendungsszenario stark variieren, wie in Kapitel 2.1 gezeigt wurde. Das zu realisierende Betriebssystem besteht dabei zunächst nur aus einer minimalen Menge von Diensten, die zum Betrieb eines Sensorknotens notwendig sind. Im laufenden Betrieb wird dann das Betriebssystem um die für eine spezielle Applikation notwendigen Dienste erweitert. Durch die lose Kopplung und die Minimierung der initial vorhandenen Dienste auf dem Knoten wird eine Flexibilität sowohl auf Applikationsebene als auch auf Betriebssystemebene erreicht.

4.3 Architektur

Ein Paradigma, welches die Anforderungen einer Abstraktion eines verteilten Systems und hoher Flexibilität erfüllt, ist das in Kapitel 2.2 aufgeführte Paradigma der Service-Orientierung. Im Rahmen des Forschungsprojektes AESOP'S Tale wurde die Anwendbarkeit des service-orientierten Paradigmas auf Sensornetze untersucht [65]. Dieses Paradigma bildet die Grundlage, nach der das in dieser Arbeit realisierte Betriebssystem für Sensorknoten *Surfer OS* vom Autor entworfen wurde.[3]

4.3.1 Grundkonzept

Dem *Surfer OS* liegen dabei zwei Ideen zu Grunde. Zum einen wird das Paradigma der Service-Orientierung im kompletten *Surfer OS* selber realisiert. Zum anderen wird die bereitgestellte Funktionalität und damit die Komplexität des *Surfer OS* auf ein Minimum reduziert.

Surfer OS bietet als Betriebssystem nicht nur eine Plattform, oberhalb derer Service-Orientierung realisiert werden kann, sondern verwendet intern Funktionalitäten ebenfalls in Form von Services. Es gilt auch (bzw. insbesondere) im *Surfer OS*-Betriebssystem das Credo: *Alles ist ein Service.* So ist beispielsweise der Zugriff auf das Funkmedium als Service realisiert. Dabei wird zwischen zwei verschiedenen Diensten, nämlich dem direkten Zugriff und dem Zugriff über den Protokollstapel (engl.: *radio stack*) unterschieden. Unter der Voraussetzung, dass Services hinzugefügt, entfernt und auch ersetzt werden können, wird ein hohes Maß an Flexibilität nicht nur auf Anwendungsebene, sondern auch im Betriebssystem selber realisiert. Am Beispiel des Dienstes für den Funkzugriff bedeutet dies, dass der gesamte Protokollstapel ersetzt werden kann. Das Betriebssystem kann durch die stringente Einhaltung dieses Paradigmas ständig an verschiedenste Szenarien und Anforderungen angepasst werden.

Um den auf den Sensorknoten geltenden Rahmenbedingungen des geringen Speichers Genüge zu tun, wird die Funktionalität des *Surfer OS* in initialen Zustand auf ein

[3] Der Name ergab sich dabei aus der phonetischen Ähnlichkeit zu dem Wort Service und der Strandnähe der Universität zu Lübeck.

Minimum reduziert. Nur Dienste, die für *Surfer OS* und seine Fähigkeit Services einzubinden notwendig sind, sind Bestandteil von *Surfer OS*. Dies bedeutet, das keinerlei Funktionalität von komplexeren Diensten wie beispielsweise Wegewahl zur Multihopkommunikation oder Lokalisation auf den Geräten vorhanden ist. Die Idee ist, dass man die Sensorknoten nur mit dem *Surfer OS*-Betriebssystem programmiert in seinem Anwendungsszenario ausbringt. Nach der Ausbringung werden komplexere Dienste, die für die Anwendung notwendig sind, auf die Geräte übertragen und in das laufende System eingebunden. Die Dienste formen in ihrer Gesamtheit die Anwendung und können auf die gleiche Weise wieder entfernt oder bei Bedarf im laufenden Betrieb ausgetauscht werden.

Dieser Ansatz bietet einen weiteren Vorteil: Die Tatsache, dass Dienste zur Laufzeit ihre Bindung an einen Knoten aufgeben und auf einen anderen Knoten migrieren können, erlaubt die Realisierung weiterer Konzepte zur Steigerung der Transparenz gegenüber dem Benutzer und zur Erhöhung der Lebensdauer und Effizienz des Sensornetzes. So muss nicht jedes Gerät über die selben Dienste verfügen. Die Softwarekonfiguration der Geräte untereinander kann sich unterscheiden und gegebenfalls an lokale Gegebenheiten anpassen, die erst zur Laufzeit bestimmt werden können. Ein Beispiel hierfür sind Algorithmen, die abstrakte Anforderungen des Benutzers an das Netz interpretieren und auf dieser Basis eine optimale Verteilung der Dienste im Netz bewirken. In Kapitel 6 wird *DySSCo* vorgestellt, ein Verfahren, das dem Programmierer einer Sensornetzanwendung ermöglicht, Anwendungen über prozentuale Abdeckungen von Diensten auf den Sensorknoten zu formulieren. Der zugrundeliegende Algorithmus realisiert selbstorganisierend die Abdeckungen auch unter sich ändernden Bedingungen, wie Knotenausfall, Mobilität oder neuen Anforderungen seitens des Benutzers. Ein weiterer Ansatz ist die Migration eines Dienstes, wenn erkannt wird, dass die Ressourcen auf dem Knoten, wie beispielsweise Energieversorgung, knapp werden. Der Dienst kann die Bindung an den Knoten aufgeben und sich auf einen anderen Knoten übertragen. So können wichtige Funktionalitäten sowie Daten und Zustände auf einem anderen Knoten weiter angeboten werden. Auf diese Möglichkeit wird in Kapitel 5 eingegangen.

4.3.2 Systemkomponenten

Um die genannten Grundideen des minimalen Betriebssystem und der durchgehenden Service-Orientierung umzusetzen, muss zunächst die Frage beantwortet werden, welche Funktionalität zwingend initial auf einem Sensorknoten vorhanden sein muss. Wie bereits beschrieben sollen im *Surfer OS*-Betriebssystem nur Dienste zur Verfügung stehen, die zur Einbindung von neuen Diensten notwendig sind. Neue Dienste werden in der Regel über Funk im Netz verteilt. Eine Ausnahme stellt hier der Knoten dar, der direkt über RS232 in Verbindung zum Rechner steht und somit die Dienste über die RS232 Schnittstelle erhält. In beiden Fällen muss das *Surfer OS*-Betriebssystem auf die Empfangsereignisse reagieren und Aktionen zur Verarbeitung der Pakete initiieren. Hierzu bedarf es eines Schedulers, der Ereignisse und Aufgaben verwaltet (*Task Management*). Zur Speicherung der empfangenen Servicepakete und letztendlich des kompletten Diens-

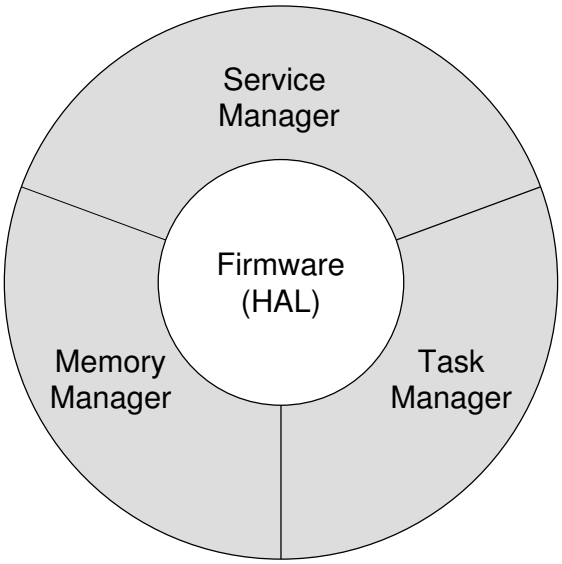


Abbildung 4.1: Komponenten des *Surfer OS*

tes ist es notwendig, dass Speicher sowohl im RAM als auch im Flash bereitgestellt wird. Daher muss im *Surfer OS* eine Komponente zur Speicherverwaltung vorhanden sein (*Memory Management*). Um die empfangenen Dienste letztendlich nutzbar zu machen und anderen Diensten zur Verfügung zu stellen, wird außerdem eine Komponente zum Einbinden und Verwalten von Diensten benötigt (*Service Management*). Mit diesen drei Komponenten zum Service Management, Memory Management und zum Task Management dargestellt in Abbildung 4.1 ist es möglich, das *Surfer OS* um weitere Dienste zu erweitern.

Service Management

Die Verwaltung der Dienste übernimmt der sogenannte *ServiceManager*. Der *ServiceManager* übernimmt dabei drei Aufgaben, die er über vordefinierte Schnittstellen zur Verfügung stellt: 1) Der *ServiceManager* bietet eine Schnittstelle zum Registrieren und Abrufen von Diensten und ihren Funktionalitäten. 2) Der *ServiceManager* koordinidert das Einbinden von Diensten auf dem Knoten. 3) Der *ServiceManager* realisiert die Verteilung von Diensten im Netz.

Alle Dienste registrieren sich mit ihren angebotenen Funktionalitäten beim *ServiceManager*. Dabei stellen die Dienste ihre Funktionalität über die Adressen der einzelnen Funktionen (*Funktionspointer*) zur Verfügung. Der *ServiceManager* speichert diese Adresse zusammen mit dem Namen des Dienstes und dem vom Dienst vorgegeben Funktionsnamen. Im *ServiceManager* sind initial alle Dienste registriert, die auf dem Knoten bereits zur Verfügung stehen. So gibt es neben Funktionen zur Speicherverwaltung, wie `malloc` und `free` im Service `memory` und dem Scheduler im Service `task` auch verschiedene Hard-

warefunktionalitäten im Service `hw_utils`, wie beispielsweise den direkten Zugriff auf die Funkschnittstelle über die Funktion `radio_send`, welche die entsprechenden Firmwarefunktionen der *Pacemate*-Firmware ruft. Weiterhin sind auch Services registriert, die noch nicht auf dem Knoten präsent sind. Dadurch werden vordefinierte Schnittstellen für grundlegende Sensornetzfunktionalitäten wie bspw. das Senden und Empfangen über Funk angeboten. So gibt es einen Service `radio`, der den Funkprotokollstapel darstellt. Die vorhandene Implementierung dieses Services greift jedoch nur auf die entsprechenden Funktionen des Services `hw_utils` zu. Ein Service, der einen Protokollstapel mit Wegewahlverfahren realisiert, muss samt des Services für das Wegewahlverfahren zunächst auf die Knoten übertragen werden. Will ein Service eine Funktionalität eines anderen Service verwenden, so bietet der *ServiceManager* eine Schnittstelle, mittels derer man die Adresse der entsprechenden Funktion erhält.

Eine zweite Aufgabe liegt in der Einbindung empfangener Services während der Laufzeit des Systems. Solche Dienste sind unabhängig vom *Surfer OS*-Betriebssystem übersetzte Stücke von Maschinencode, die in das *Surfer OS*integriert werden. Dabei muss der *ServiceManager* zunächst entscheiden, ob der empfangene Service eingebunden wird. Anschließend muss der Prozess der Migration, der im folgenden Kapitel 5 detailiert beschrieben wird, auf dem Knoten abgeschlossen werden.

Die dritte Aufgabe des *ServiceManagers* besteht darin, es zu ermöglichen, dass Services im ganzen Sensornetzwerk verteilt werden können. Die Dienste werden dabei je nach Größe in mehreren sogenannten *Servicepaketen* versandt. Hierfür muss ein einfaches Protokoll zur Verfügung stehen, welches dem *ServiceManager* erlaubt, Servicepakete gesondert zu behandeln, zu verarbeiten und gegebenfalls weiterzuleiten.

Memory Management

Die Speicherverwaltung wird vom sogenannten *MemoryManager* übernommen. Der *MemoryManager* teilt sich dabei in zwei Komponenten auf. Die erste Komponente verwaltet das RAM. Dies beinhaltet den Stapelspeicher (engl.: *Stack*) und den dynamischen Speicher (engl.: *Heap*). Die zweite Komponente realisiert den Zugriff auf den Flash-Speicher.

Für die Verwaltung des RAMs ist inbesondere die Realisierung der dynamischen Speicherallokation von Bedeutung. Es muss eine Implementierung gefunden werden, die den Speicher optimal ausnutzt und effizient verwaltet. Gleichzeitig muss die Speicherverwaltung dafür Sorge tragen, dass der Heap nicht Speicherbereiche überschreibt, die bereits vom Stack genutzt werden.

Die Komponente zur Verwaltung des Flash-Speichers muss persistente Daten, die über einen Neustart hinaus erhalten bleiben sollen, im Speicher ablegen können und gegebenenfalls wieder laden können. Solche Daten könnnen beispielsweise empfangener Servicecode sein. Die Verwaltungskomponente muss dabei die Unterteilung des Flash-Speichers in verschiedene Sektoren wie in Kapitel 3.3 beschrieben berücksichtigen. Hier ist insbesondere zu beachten, dass der Flash nicht problemlos überschrieben werden kann, sondern lediglich Bits mit dem Wert 1 auf 0 gesetzt werden können. Um also Daten zu

überschreiben, muss zunächst der gesamte betroffene Sektor gelöscht (auf 1 gesetzt) werden, bevor ein altes Datum mit einem neuen Datum überschrieben werden kann. Eine weitere Herausforderung liegt dabei darin, dass die Informationen über die Belegung der Flash-Speicher-Datenstruktur ebenfalls persistent gehalten werden müssen. Nur so kann gewährleistet werden, dass nach einem Neustart, ausschließlich freie Speicherbereiche im Flash-Speicher allokiert werden und nicht fälschlicherweise auf bereits beschriebenen Speicher zugegriffen wird. Die belegten Speicherbereiche müssen zudem wieder den Programmmodulen (bzw. Diensten) zugewiesen werden können. So müssen Daten, die von einem Dienst persistent in den Flash-Speicher abgelegt wurden, diesem wieder zur Verfügung gestellt werden, da dem Dienst nach einem Neustart die Existenz dieser Daten unbekannt ist. Das bedeutet, dass zunächst identifiziert werden muss, zu welchem Dienst einzelne belegte Speicherbereiche gehören. Danach kann die Referenz auf diesen Speicherbereich dem Dienst übergeben werden.

Task Management

Bei der Planung von Aufgaben unterteilt sich der sogenannte *TaskManager* in zwei Module: den *EventDispatcher* und den eigentlichen *TaskManager* selber.

Der *EventDispatcher* benachrichtigt registrierte Services über verschiedene Ereignisse, die sowohl seitens der Hardware als auch seitens der Software erzeugt werden. Zu solchen Ereignissen zählt beispielsweise das Empfangen einer Funknachricht oder das Drücken einer der drei Tasten des *Pacemates*. Durch diese Ereignisse wird der Programmfluss (die Hintergrundaufgabe) unterbrochen (engl.: *interrupt*). Der *EventDispatcher* bietet daber Schnittstellen, bei der die einzelnen Services sich für bestimmte Ereignisse registrieren können und die Adresse einer Rückruffunktion (engl.: *callback function*) hinterlegen.

Der eigentliche *TaskManager* regelt den sequentiellen Ablauf von Hintergrundaufgaben. Dienste können hier einzelne Aufgaben (ebenfalls über Rückruffunktionen) für bestimmte Zeitpunkte registrieren. Sobald der Zeitpunkt erreicht ist, wird die Rückruffunktion vom *TaskManager* aufgerufen.

4.4 Implementierung

Im Folgenden wird beschrieben, wie die einzelnen Komponenten zum Service Management, Memory Management und Task Management innerhalb des *Surfer OS* realisiert sind.

4.4.1 Service Management

Der *ServiceManager* muss wie beschrieben die drei Aufgaben der Registrierung und Bereitstellung, der Integration und der Verteilung von Diensten implementieren.

Registrierung und Bereitstellung

Zur Registrierung eines Dienstes bietet der *ServiceManager* eine Schnittstelle, über die ein Service unter Angabe eines global eindeutigen Servicenamens und einer eindeutigen Serviceidentifikationsnummer, die beide den Typ des Dienstes identifizieren (beispielsweise: `Routing`), angemeldet wird. Die Signatur dieser Funktion innerhalb des *ServiceManager* ist in Auflistung 4.1 angegeben. Dem neuen Dienst wird dabei eine lokale Identifikationsnummer zugewiesen, welche die Serviceinstanz identifiziert. Die Identifikation über den Namen dient zum Auffinden und Verwenden des Dienstes durch andere Dienste. So möchte der Programmierer beispielsweise den `Radio`-Service zum Versenden von Funknachrichten auch mittels dieses Namens auffinden und verwenden können. Die zusätzliche numerische Identifikation dient zur speichereffizienten internen Verarbeitung. So werden intern die einzelnen vom Dienst angebotenen Funktionen mittels der lokalen Identifikationsnummer dem Dienst zugeordnet.

```
720 uint16 ServiceManager_registerReplaceableService(char*
      service_name, uint32 global_service_id)
721 {
722   if (service_exists(service_name) == true)
723     return SERVICE_MANAGER_INVALID_SERVICE_ID;

725     uint16 local_service_id = generate_local_service_id();
726   [...]

740   return local_service_id;
741 }
```

Quelltext 4.1: Schnittstelle des *ServiceManagers* zur Registrierung eines neuen Dienstes

Zur Registrierung der einzelnen Funktionen bietet der *ServiceManager* eine Schnittstelle angegeben in Auflistung 4.2, die unter Angabe der lokalen Serviceidentifikationsnummer, dem Funktionsnamen und der Adresse der zu registrierenden Funktion (Funktionspointer, engl.: *function pointer*) Einträge in die Liste der zur Verfügung stehenden Funktionen für den Dienst hinzufügt.

In gleicher Weise bietet der *ServieManager* auch die entsprechenden Funktionen zum Entfernen der Dienste und Funktionen.

Um Funktionalitäten von Diensten nutzen zu können, bietet der *ServiceManager* eine Schnittstelle zum Abrufen der Funktionsadressen mittels Servicename und Funktionsnamen. Die Schnittstelle liefert daraufhin die Adresse der Funktion zurück, die seitens des Benutzers auf eine Funktionsadresse mit korrekter Signatur umgewandelt (engl.: *type casting*) werden muss. Dies bedeutet, dass die Typen der Übergabe- und Rückgabeparameter für diese Adresse benannt werden müssen. Durch Zuweisung dieser Adresse auf

```
667 bool ServiceManager_RegisterFunction(uint16 local_service_id,
        char* fun_name, fun_ptr fun_pointer)
668 {
669     ServiceManager_FunRegEntry* new_entry = malloc(sizeof(
            ServiceManager_FunRegEntry));
670   if (new_entry == NULL)
671     return false;
672   [...]

690     counted_functions_++;
691   return true;
692 }
```

Quelltext 4.2: Schnittstelle des *ServiceManagers* zur Registrierung von Funktionen eines Dienstes

eine lokale Variable von Typ dieses Funktionspointers kann die abgerufene Funktion nun vom Benutzer durch Aufrufen der Variablen verwendet werden.

Durch diese Umsetzung können Dienste über den *ServiceManager* eigene Funktionen anbieten und Funktionen anderer Dienste nutzen.

Integration

Nachdem beschrieben wurde, wie einzelne Dienste sich im *ServiceManager* registrieren, muss geklärt werden, wie die Integration eines Dienstes abläuft. Die Integration ist in die folgenden Einzelschritte unterteilt: 1) Allokation von Speicher, 2) Relokation des Servicecodes, 3) Initialisieren des Dienstes und 4) Registrieren des Dienstes.

Im ersten Schritt wird überprüft, ob auf dem Sensorknoten genügend Speicherplatz für den Dienst zur Verfügung steht. Dabei muss zum einen sichergestellt werden, dass im Programmspeicher, dem Flash, genügend Platz für den gesamten Dienst zur Verfügung steht. Die Variablen des Dienstes müssen zusätzlich im RAM platziert werden, damit auf diese während der Ausführung des Dienstes nicht nur lesend, sondern vor allem auch schreibend zugegriffen werden kann. Daher muss zudem zugesichert sein, dass für die Variablen in dem DATA- und BSS-Segment des Codes[4] genügend Speicherplatz im RAM (bzw. auf dem Heap) vorhanden ist. Der benötigte Speicher wird daraufhin bereits allokiert.

Basierend auf den Adressen des allokierten Speichers werden im zweiten Schritt der Programmcode und die Variablen des Dienstes neu platziert (relokiert) und angepasst. Auf diesen Vorgang wird im folgenden Kapitel 5 im Detail eingegangen. Anschließend wird der angepasste Dienst in den Flash-Speicher geschrieben und ist persistent.

[4] Auf die Segmente eines Programms wird in Kapitel 5 im Einzelnen eingegangen.

Damit der zu integrierende Dienst Informationen mit dem *ServiceManager* austauschen und seine öffentlichen Funktionen registrieren kann, ruft der *ServiceManager* die für jeden Dienst im *Surfer OS* vorgegebene `ServiceInit`-Funktion mit gegebener Signatur. Durch die Erstellung des Dienstes gemäß dem in Kapitel 5 beschriebenen Verfahren kann erwirkt werden, dass diese Funktion sich bei jedem Dienst immer direkt am Anfang des Programmcodes befindet. Durch dieses Verfahren ist der Informationsaustausch zwischen *Surfer OS* und dem Dienst realisiert. Der Dienst erhält unter anderem Funktionspointer zu den Schnittstellen des *ServiceManagers*. Durch diese Schnittstellen kann der Dienst eigene Funktionen beim *ServiceManager* registrieren und präsente Dienste auf dem Knoten nutzen.

Abschließend führt der *ServiceManager* die Registrierung des Dienstes durch. Dies bedeutet, dass der empfangene Dienst in eine Liste mit allen auf den Knoten zur Verfügung stehenden Diensten eingefügt wird. Damit ist die Integration eines Dienstes abgeschlossen.

Um einen Dienst zu entfernen, werden die entsprechenden Speicherbereiche freigegeben sowie die Einträge im *ServiceManager* entfernt. Jeder Dienst implementiert daher eine `ServiceStop`- und `ServiceCleanUp`-Funktion. Diese sind beim *ServiceManager* registriert. Sie entfernen bei Aufruf die registrierten Rückruffunktionen und geben den dynamisch belegten Speicher des Dienstes wieder frei. Somit kann der Dienst angehalten und gegebenenfalls wieder komplett von einem Knoten entfernt werden.

Beim Ersetzen eines Dienstes wird zunächst die alte Version entfernt und dann der neue Dienst wie oben beschrieben auf dem Gerät integriert. Die Entscheidung, ob ein Dienst ersetzt wird, liegt an dem Vergleich einer Versionsnummer der Dienste. Stimmt diese nicht überein, so wird der auf dem Knoten vorhandene Dienst mit dem neu empfangenen Dienst ersetzt.

Verteilung

Die Migration von Diensten erlaubt die Veränderung und Adaption der Sensornetzanwendung und realisiert damit administrative Prozesse, die von der eigentlichen Applikation getrennt sein müssen. Servicepakete und deren Verteilung müssen somit für die Anwendung nicht sichtbar sein. *Surfer OS* verwendet dazu ein spezielles Paketformat, welches beim Empfangen erkannt wird und somit lediglich an den *ServiceManager* ausgeliefert wird. Dieses Paketformat beinhaltet eine eigene Adressierung auf dem RS232- sowie auf dem Funkmedium. Dabei wird zwischen 1-hop- und Multi-hop-Versenden unterschieden. Empfängt ein Knoten Servicepakete über RS232, die nicht für ihn oder an alle Knoten adressiert sind, so leitet er diese über Funk weiter. Empfängt ein Knoten solche Pakete über Funk, so werden diese nur weitergeleitet, wenn ein spezielles Multihop-Flag im Paketformat gesetzt ist. Dabei kommt als Wegewahlprotokoll ein einfaches Fluten zum Einsatz. Die Angabe von Serviceidentifikationsnummer und Byte-Versatz ermöglicht letztendlich die Zuordnung einzelner Servicefragmente (Pakete) zu einem gesamten Service auf dem Zielknoten.

4.4.2 Memory Management

Zur Realisierung der Datenstrukturen im Heap und im Flash werden verkettete Listen verwendet. Ein Listenelement repräsentiert dabei einen allokierten Speicherbereich. Die Informationen über die Listenelemente umfassen die Größe und die Adresse des nächsten Listenelements und sind dabei den jeweiligen Speicherbereichen vorangestellt, wie in Abbildung 4.2 dargestellt. Dies erlaubt eine kompakte und flexible Implementierung, da die Anzahl der dynamisch erzeugbaren Elemente lediglich vom zur Verfügung stehenden Speicherplatz, nicht jedoch durch die zu Grunde liegende Datenstruktur begrenzt ist. Wird jedoch durch fehlerhafte Implementierungen über die Begrenzungen dieser Speicherbereiche geschrieben, erfolgt das Überschreiben der Informationen über die Datenstruktur, wodurch die folgende Allokation von Speicher fehlerhaft wird. Anhand der Informationen über Größe das aktuellen Elementes und die Adresse des nächsten Elementes lassen sowohl die belegten Bereiche, wie auch die freien Bereiche berechnen. In der Implementation des dynamischen Speichers im RAM verwendet *Surfer OS* eine *Best-Fit*-Strategie. Bei dieser Speicherbelegungsstrategie wird einer neuer Speicherbereich dort belegt, wo nach der Belegung zwischen dem neuen und dem nächsten Element der kleinste Speicherbereich ungenutzt bleibt.

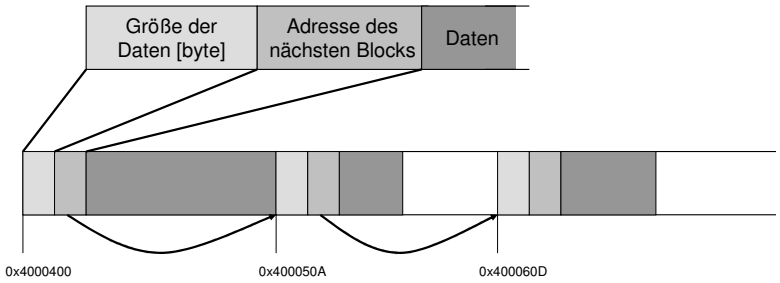


Abbildung 4.2: Speicherdatenstruktur über verkettete Liste

In der Implementierung des Flash-Speichers hält jedes Lsitenelement zudem Informationen über den Besitzer des Speicherbereiches. So können die persistenten Daten nach einem Neustart wieder dem besitzenden Modul zugeordnet werden. Dies kommt insbesondere im Fall der Dienste zum Einsatz. So können Speicherbereiche, in denen der *ServiceManager* als Besitzer registriert ist, diesem wieder zugeordnet werden. Dienste, die in dem Flash gespeichert sind, können so nach dem Neustart dem *ServiceManager* wieder übergeben und von diesem eingebunden werden.

Um der Eigenschaft, den Flash-Speicher nicht einfach überschreiben zu können, Genüge zu tun, wird der letzte Sektor des Flash-Speichers als Austauschsektor verwendet. Da sich beim Einfügen in die Datenstruktur nicht nur der Speicherinhalt selbst, sondern insbesondere die in der Datenstruktur gehaltenen Informationen ändern, wird der Inhalt des betroffenen Sektors zunächst in den zuvor gelöschten Austauschsektor geschrieben. Bei diesem Vorgang wird der betroffene Sektor in Blöcken von 256 Bytes in den RAM geladen. Dabei werden die Informationen in der Datenstruktur aktualisiert und die zu

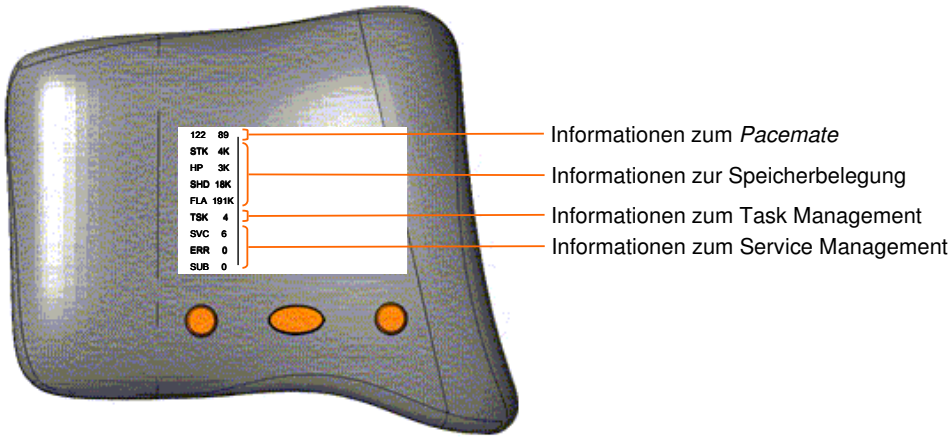


Abbildung 4.3: Schematischer *Pacemate* mit Anzeige des *Surfer OS*

speichernden Daten ergänzt. Der modifizierte Block wird in den Austauschsektor geschrieben bevor der nächste Block gelesen wird. Anschließend wird der betroffene Sektor gelöscht und die Daten aus dem Austauschsektor wieder zurückgeschrieben.

4.4.3 Task Management

Der *EventDispatcher*, der beim Auftreten von Ereignissen Rückruffunktionen von Diensten aufruft, ermöglicht den Diensten, diese Funktionen für verschiedene Kategorien von Ereignissen zu registrieren. So kann ein Dienst eine Rückruffunktion für Funkereignisse, Tastenereignisse o. ä. registrieren. Der *EventDispatcher* realisiert für jede Kategorie eine Liste von Rückruffunktionen, die beim Auftreten des Ereignisses sequentiell aufgerufen werden.

Der *TaskManager*, welcher das Registrieren für Hintergrundaufgaben realisiert, hält diese Aufgaben ebenfalls in einer Liste. Diese ist jedoch nach den registrierten Zeitintervallen sortiert. So steht die Aufgabe, die als nächstes bearbeitet werden soll, am Anfang der Liste. Die Liste wird parallel zu der in Kapitel 3.4.1 beschriebenen `SYS_Loop` überprüft, ob der Zeitstempel für die Ausführung der nächsten Aufgabe erreicht ist.

4.5 Beispiel

Das Betriebssystem *Surfer OS* bietet nur minimale Funktionalität. Um jedoch *Surfer OS* weiterzuentwickeln und gegebenenfalls Programmierfehler identifizieren zu können, wurden bereits Dienste in *Surfer OS* integriert, die Informationen über den Status des betreffenden *Pacemate*-Sensorknoten geben. *Surfer OS* verfügt dabei über verschiedene Anzeigen, die in einem festgelegten Intervall wechseln. Dabei werden unter anderem auch Informationen über den Kommunikationsstatus, beispielsweise empfangene und

gesendete Funkpakete, aufgeführt. Abbildung 4.3 zeigt beispielhaft eine Anzeige eines *Pacemate*-Sensorknotens mit *Surfer OS*. Die Darstellung zeigt dabei die Angaben über die *Pacemate*-Hardware, die Speicherbelegung (Memory Management), das Task Management sowie das Service Management.

Die Angaben über die *Pacemate*-Hardware beinhalten die Seriennummer des Gerätes, die gleichermaßen die Adresse des Gerätes für Funkpakete darstellt. Außerdem wird der aktuelle Ladestand des Akkumulators angezeigt.

Die Informationen über die Speicherbelegung zeigen, wie groß der Stapelspeicher (hier STK für Stack) ist. Zudem wird die Menge des belegten dynamischen Speichers (HP für Heap) angegeben. Ein häufiger Fehlerfall ist, dass der Stapelspeicher in den dynamischen Speicher wächst und dort Daten überschreibt. Dafür wird die Entfernung zwischen dem Ende des Stapelspeichers und dem Ende des am nächsten zum Stack allokierten dynamischen Speichers angegeben (SHD für Stack-Heap-Distanz). Neben der Speicherbelegung im RAM wird außerdem der freie Speicherplatz im Flash-Speicher (FLA für Flash) für neue Dienste angezeigt.

Die Informationen zum Task Management zeigen, wie viele Hintergrundaufgaben (TSK für Tasks) auf dem Gerät aktuell registriert sind.

Die Informationen zum Service Management geben an, wie viele Dienste (SVC für Service) aktuell auf dem Gerät präsent sind. Zudem wird ein Fehlercode (ERR für Error) angezeigt, der bei dem letzten Empfang eines Dienstes erzeugt wurde. Hierdurch kann man Rückschluss auf etwaige Probleme beim Einbinden eines neuen Dienstes erhalten. Außerdem werden die Abhängigkeiten zwischen Diensten (SUB für Subscriptions) angezeigt.

Die Informationen über den Status des einzelnen Sensorknoten helfen bei der Weiterentwicklung von *Surfer OS* sowie bei der Erstellung neuer Dienste, insbesondere solcher, die verteilte Algorithmen und Protokolle implementieren. Anhand der angezeigten Daten lassen sich leicht etwaige Skalierungsprobleme, die sich beispielsweise durch die Auslastung des Speichers sowie das Kommunikationsaufkommen auszeichnen, erkennen. Die Anzeige des *Pacemate* dient somit hierbei als Fenster in den internen Status des *Surfer OS*.

4.6 Ergebnis

In diesem Kapitel wurden verschiedene existierende Sensorknotenbetriebssysteme vorgestellt. Diese wurden unter dem Aspekt der Applikationserstellung betrachtet. Besonderes Augenmerk wurde dabei auf die Flexibilität und die Benutzerabstraktion gelegt, die die Betriebssysteme bieten. Dabei wurde auch auf Middlewareansätze eingegangen, die aufbauend auf dem zu Grunde liegenden Betriebssystem als Ziel haben, von dem Sensornetz als verteiltem System zu abstrahieren und dem Benutzer eine intuitive Sicht auf das Netz zu vermitteln. Es wurde die Anforderung der Anpassbarkeit während des Betriebs des Sensornetzes formuliert. Diese Anpassbarkeit soll sowohl für die Sensornetzanwendung

aber auch für tiefere Schichten, wie beispielsweise dem Funkprotokollstapel, gewährleistet sein. Dabei soll dem Benutzer der Sensornetztechnologie, bzw. dem Programmierer eine einfache Abstraktion bei größtmöglichem Freiheitsgrad bei dem Entwurf seiner Anwendung geboten werden.

Basierend auf der Erkenntnis, dass weder die existierenden Betriebssysteme noch die Middlewareansätze diese Anforderungen zur Genüge erfüllen, hat der Autor das *Surfer OS* entwickelt und vorgestellt. Das *Surfer OS* realisiert dabei das Paradigma der Service-Orientierung auf einem eingebettetem System, welches bereits in der Entwicklung von verteilten Anwendungen im Bereich der Geschäftsanwendungen verbreitet ist. *Surfer OS* ist auf ein Minimum reduziert und bietet dabei als Betriebssystem initial nur die Hardwareabstraktionsschicht, ein Task Management, Memory Management und die Möglichkeit, Dienste dynamisch zur Laufzeit einzubinden und zu verteilen.

In *Surfer OS* wird alles als Service aufgefasst. Dieser Ansatz ermöglicht, dass sogar Komponenten wie der Funkprotokollstapel (und somit beispielsweise das Wegewahlverfahren) im laufenden Betrieb ersetzt werden können. Dadurch kann man unter anderem den in Kapitel 2.1.3 aufgeführten Herausforderungen einer Vielzahl von Tests am Ort der Ausbringung des Sensornetzes Herr werden, indem man vor Ort beispielsweise das passende Wegewahl- oder Lokalisationsverfahren auswählt und als Service auf die Geräte überträgt.

Weiterhin ist es in *Surfer OS* nicht erforderlich, dass jedes Gerät die gleiche Softwarekonfiguration hat bzw. die gleichen Dienste anbietet. Insbesondere basierend auf lokalen Gegebenheiten kann hier nach der Ausbringung zur Laufzeit festgelegt werden, welcher Knoten welche Dienste anbietet. Hierbei können auch selbstorganisierende Ansätze wie zum Beispiel das in Kapitel 6 aufgeführte *DySSCo*-Protokoll zum Einsatz kommen. Weiterhin können Dienste ihre Bindung zum Gerät aufgeben, wenn beispielsweise der Speicher auf dem aktuellen Gerät zu knapp wird oder die Energieressourcen schwinden.

Mittels der im folgenden Kapitel 5 vorgestellten Schnittstelle für die Implementierung einzelner Dienste können Anwender leicht Dienste für *Surfer OS* implementieren. Durch die Abstraktion der Service-Orientierung lassen sich so einzelne Module leicht erstellen und wiederverwenden. Applikationen in *Surfer OS* werden somit durch ein Zusammensetzen bzw. Komponieren einzelner Dienste realisiert. *Surfer OS* bildet somit die Grundkomponente der in dieser Arbeit realisierten service-orientierten Infrastruktur.

5 Multi-hop-Migration von Programmcode

Im vorigen Kapitel wurde das *Surfer OS* als service-orientiertes Betriebssystem für Sensorknoten vorgestellt. Eine Hauptcharakteristik des *Surfer OS* besteht in der Fähigkeit, Dienste während der Laufzeit migrieren und integrieren zu können. Dies impliziert, dass die Dienste unabhängig vom eigentlichen System implementiert werden. Die Herausforderung besteht darin, dass den Diensten die Funktionalitäten des Knoten, dessen Schnittstellen und die Speicheradressen der Schnittstellen zum Zeitpunkt der Erstellung gänzlich unbekannt sind. Zudem ist unbekannt an welchen Speicherstellen ein Dienst auf dem Knoten abgelegt wird. Der Aufruf eines Dienstes seitens des Betriebssystems bzw. das Nutzen von Funktionalitäten des Knotens seitens eines Dienstes ist daher nicht trivial. Dieses Problem ist in der Informatik bekannt als das Relokationsproblem [100].

In diesem Kapitel wird auf die technischen Details der Migration eingegangen. Es wird das Verfahren beschrieben, das der Autor innerhalb des *Surfer OS* realisiert hat. Dieses Verfahren ist jedoch auch auf andere Systeme und Plattformen übertragbar. Dabei wird zunächst auf bestehende Ansätze zur Migration von Diensten eingegangen. Es werden die technischen Details von Verfahren vorgestellt, die im vorigen Kapitel im Rahmen der Vorstellung verschiedener Betriebssysteme und Middlewares nur oberflächlich angerissen worden sind. Verfahren, die nur ein vollständiges Ersetzen des Sensorknotenprogramms erlauben, werden dabei nicht berücksichtigt. Danach werden die Anforderungen an die Migration formuliert und die verwandten Arbeiten dahingehend geprüft. Im Rahmen der Architektur wird zunächst auf die Grundlagen des allgemeinen Codeerstellungsprozesses (engl.: *build process*) eingegangen. Anschließend wird ein neuer Ansatz für die Migration von Diensten in Sensornetzen beschrieben. Darauffolgend werden im Abschnitt Implementation die technischen Details der im Rahmen dieser Arbeit umgesetzten Realisation der Migration beschrieben. Das Kapitel schließt mit einer Bewertung der Ergebnisse.

5.1 Verwandte Arbeiten

Bereits im vorangegangen Kapitel ist auf Betriebssysteme und Middlewareansätze eingegangen worden, die dem Programmierer die Flexibilität bieten, Programmmodule dynamisch in eine laufende Sensornetzapplikation einzubinden. Im Folgenden werden die technischen Details der Migration in den verwandten Arbeiten, die eine dynamische Reprogrammierung im laufenden Betrieb ermöglichen, näher beschrieben.

5.1.1 Virtuelle Maschine mit Maté

Im Rahmen der Middlewareansätze wurde das Prinzip der virtuellen Maschine genannt, welches wie aufgeführt mit Maté auf Basis des TinyOS Betriebssystems realisiert wurde. Eine virtuelle Maschine stellt dabei eine in Software realisierte Maschine dar. Die Migration von Programmcode für virtuelle Maschinen wird dadurch realisiert, dass der Code, der in der speziellen Syntax vorliegt, auf dem Knoten von einem sogenannten *Interpreter* interpretiert wird. Die virtuelle Maschine abstrahiert dabei so von der eigentlichen Hardware, dass die Speicheradressen, an denen sich der Programmcode physisch befindet, keine Rolle spielen. Das oben genannte Relokationsproblem wird dadurch umgangen.

5.1.2 Code Relokation in Contiki

Alternativ zu dem Ansatz der virtuellen Maschine kann der zu migrierende Programmcode übersetzt und der resultierende Maschinencode übertragen werden. Dies erspart die virtuelle Maschine und den Interpreter auf dem Knoten. Bei der Übersetzung von Programmcode in Maschinencode wird eine sogenannte Objektdatei erzeugt. Abhängig von der Zielpattform kann diese Datei in unterschiedlichen Formaten vorliegen. Am weitesten verbreitet ist dabei das *Executable Linking Format* (ELF). Das ELF-Format ist vom *Tool Interface Standard Comitee* (TIS Comitee) im Rahmen des Tool Interface Standard 1995 als Standard für ausführbare Programme anerkannt worden [106]. Neben dem eigentlichen Maschinencode enthält die ELF-Datei unter anderem auch eine sogenannte Symboltabelle. Durch sogenanntes *dynamisches Binden* (engl.: *dynamic linking*) enthält die ELF-Datei alle Symbole (Variablen und Funktionsnamen), die zum Zeitpunkt des Übersetzens und Bindens des Programms nicht aufgelöst werden konnten.

Das Contiki Betriebssystem nutzt genau diese Informationen aus. Contiki implementiert einen *Linker*, der das ELF-Format lesen kann und die fehlenden Symbole (beispielsweise Aufrufe von Betriebssystemfunktionen) auflöst sowie den Programmcode des Moduls neu platziert und ausführt. Das ELF-Format erzeugt jedoch hohe Kosten (engl.: *overhead*) in Form von Bytes, die übertragen werden müssen. Dies ist darin begründet, dass das ELF-Format neben anderen Informationen auch alle Symbolnamen der referenzierten Funktionen und Variablen in einer Symboltabelle hält, die zur Laufzeit aufgelöst werden müssen. Zusätzlich ist das ELF-Format für Hardwareplattformen standardisiert, die eine Befehlsbreite von 32 oder 64 bit nutzen. Dunkels et al. schlagen daher eine kompakte Darstellung des ELF-Formats vor. Das *Compact ELF* (CELF) nutzt dabei die Tatsache, dass die Zieplattform eine 8 oder 16 bit Plattform ist. Durch diesen Ansatz kann die Größe der ELF-Datei verringert werden.

5.1.3 Positionsunabhängiger Code in SOS

Positionsunabhängiger Code (engl.: *position independant code*) ist Programmcode, der an beliebigen Speicheradressen ausgeführt werden kann. Anfangs mussten Programme immer an festen Speicheradressen ausgeführt werden. Das Prinzip des positionsu-

nabhängigen Codes wurde eingeführt, um mehrere Programme gleichzeitig auf einem Rechner laufen zu lassen. Mit der Einführung einer Speicherverwaltungseinheit (engl.: *Memory Management Unit* (MMU)), die eine dynamische Adresszuweisung realisiert, ist dieses Prinzip obsolet geworden. Eine Realisierung dynamischer Speicherverwaltung ist für eingebettete Systeme zu aufwendig. Das aufgeführte Betriebssystem SOS verwendet positionsunabhängigen Code, um einzelne Programmmodule zwischen den Knoten migrieren zu können.

5.2 Anforderungen

Das im vorigen Kapitel eingeführte Sensorknotenbetriebssystem *Surfer OS* stellt klare Anforderungen an die einzelnen Dienste. So sollen Dienste die Funktionalitäten anderer Dienste nutzen können. Die Kommunikation zwischen den verschiedenen Diensten muss also möglich sein. Desweiteren muss es möglich sein, die Flexibilität von *Surfer OS* in der Form zu unterstützen, dass sogar vom Betriebssystem selbst angebotene Dienste wie beispielsweise der Funkprotokollstapel ersetzt werden können. Die Grenzen zwischen migrierten Diensten und dem eigentlichen *Surfer OS* sind dabei fließend. Die realisierte Lösung soll dabei möglichst unabhängig von der zugrundeliegenden Hardwareplattform sein. Die aufgeführten Ansätze erscheinen daher nicht anwendbar für die Servicemigration in *Surfer OS*.

Eine virtuelle Maschine bietet durch die klare Abgrenzung zum Betriebssystem den Vorteil der Robustheit. Code kann nur in den von der virtuellen Maschine zugelassenen Grenzen operieren – das bereits genannte Sandbox-Prinzip. Dadurch lassen sich bspw. unerlaubte Speicherzugriffe abfangen und die Funktionalität des Knotens kann weiterhin gewährleistet werden.

Der Programmcode, der für die virtuelle Maschine geschrieben worden ist, muss jedoch immer interpretiert werden. Die Syntax wird bei jedem Aufruf in den eigentlichen Maschinencode der Sensorknotenplattform überführt. Zudem benötigt die virtuelle Maschine zusammen mit dem sogenannten *Interpreter* zusätzlichen Speicherplatz auf dem Sensorknoten. Die Implementation von Maté benötigt zusammen mit den (für Maté) notwendigen TinyOS Komponenten insgesamt 16 kbyte Programmspeicher und 1 kbyte RAM [55].

Desweiteren ist es durch den Ansatz der virtuellen Maschine nur schwer möglich, die vom System bereits angebotenen Dienste zu ersetzen. Dies ist in der klaren Abgrenzung von Betriebssystem und virtueller Maschine begründet. Die geforderte Flexibilität kann auf diese Weise nicht erreicht werden.

Die Verwendung eines kompakten ELF-Formates CELF in Contiki basiert auf der Annahme, dass die zugrundeliegende Architektur eine Befehlsbreite von 8 bit hat. Die *Pacemate* Plattform ist jedoch eine 32 bit Architektur. Die Kompression des ELF-Formates ist daher plattformabhängig und nicht im *Surfer OS* auf den *Pacemates* nutzbar.

Die Verwendung von positionsunabhängigem Code wie in SOS setzt voraus, dass dies von dem Übersetzer (engl.: *compiler*) unterstützt wird. Gemäß der Dokumentation der *Free Software Foundation* (FSF) für die *GNU Compiler Collection* (GCC) [23] ist diese Unterstützung jedoch nur für spezielle Architekturen verfügbar. Daher eignet sich dieser Ansatz ebenfalls nicht für die Verwendung im *Surfer OS* auf den *Pacemates*.

5.3 Grundlagen der Codeerstellung

Um den Migrationsprozess zu realisieren, ist es notwendig, dass zunächst auf die Grundlagen der Codeerstellung eingegangen wird. Damit aus einem Programm (bzw. Service), welches in einer höheren Programmiersprache wie beispielsweise C geschrieben worden ist, ausführbarer Maschinencode entsteht, kommen mehrere einzelne Programme zum Einsatz: der *Codegenerator*, der *Assembler* und der *Bindelader* (engl.: *linker*). Im Folgenden werden die einzelnen Prozesse im Detail betrachtet.

5.3.1 Codegenerierungsprozess

Nachdem das Programm vorverarbeitet und syntaktisch sowie semantisch überprüft worden ist, beginnt der Prozess der Codegenerierung. Der Prozess der Codegenerierung wird in der Regel in zwei Phasen aufgeteilt: in die *Analysephase* (engl.: *frontend*) und in die *Synthesephase* (engl.: *backend*). Die Analysephase erzeugt dabei zunächst einen sogenannten *Zwischencode* (engl.: *intermediate code*). In der Synthesephase wird Code für die Zielplattform generiert. Diese Zweiteilung erlaubt dabei das schnellere Erstellen von Übersetzern für andere Zielplattformen, da lediglich das Werkzeug der Synthesephase ausgetauscht werden muss. Zudem können plattformunabhängige Codeoptimierer, die auf dem Zwischencode arbeiten, einfach eingebunden werden.

Die Eingabe für die Synthesephase ist dabei der Zwischencode und die *Symboltabelle*, welche benötigt wird, um die Laufzeitadressen von Variablen und Funktionen aufzulösen. Ein Eintrag in die Symboltabelle wird immer dann erzeugt, wenn die Rolle eines im Programmcode vorkommenden Symbols bzw. Bezeichners festgestellt werden kann.

Mit diesen Informationen kann in der Synthesephase *Maschinencode* oder *Assemblercode* erstellt werden. Maschinencode bzw. Maschinensprache besteht aus numerischen Darstellungen (Bitfolgen) von Befehlen (*Opcodes*) und deren Argumenten, die vom Prozessor der Zielplattform direkt verarbeitet werden können, und ist für den Menschen nur schwer lesbar. Die Assemblersprache stellt demgegenüber diese Bitfolgen als menschenlesbare und merkbare Kürzel – als sogenannten *mnemonischen Symbole* (engl.: *mnemonics*, griech.: *mnēmoniká*, „Gedächtnis“) dar.

5.3.2 Codeübersetzungsprozess

Im Codeübersetzungsprozess überführt der sogenannte *Assembler* ein Programm, welches in der Assemblersprache vorliegt, in Maschinencode. Die Ausgabe des Assemblers

ist eine sogenannte *Objektdatei*, die in verschiedenen Formaten vorliegen kann. Wie bereits erwähnt ist das ELF-Format dabei eins der verbreitetsten dieser Formate. Die verschiedenen Formate haben dabei gemein, dass Informationen bzw. Programmteile eines speziellen Typs speziellen *Segmenten* zugeordnet werden. Oft wird in diesem Zusammenhang der Begriff *Sektion* (engl.: *section*) als Synonym verwendet. Eine Sektion beschreibt jedoch die kleinste Einheit der Unterteilung innerhalb einer ELF-Datei. Ein Segment kann daher aus verschiedenen Sektionen bestehen. Die drei Hauptsegmente sind dabei: das TEXT-Segment, das DATA-Segment und das BSS-Segment.

Das TEXT-Segment beinhaltet die Programmanweisungen. Die Anweisungen ändern sich in der Regel nicht zur Laufzeit. Daher ändert sich dieses Segment weder in Größe noch in Inhalt und kann somit speziell behandelt werden. Manche Plattformen können diesem Segment spezielle Rechte zuweisen, wie beispielsweise das Segment nur für Lesezugriffe (engl.: *read-only*) freizugeben.

Das DATA-Segment beinhaltet die vorinitialisierten statischen und globalen Variablen des Programms mit ihren initialen Werten. Im Gegensatz zu dem TEXT-Segment können die Inhalte im DATA-Segment während der Laufzeit ihre Werte verändern. Daher muss das DATA-Segment anders als das TEXT-Segment behandelt werden.

Das BSS-Segment (*Block Started by Symbol*) ist speziell für uninitialisierte Daten. Uninitialisierte Daten in ein eigenes Segment zu platzieren, ermöglicht die Einsparung von Speicher in der resultierenden Objektdatei. So muss lediglich die Größe des BSS-Segments gespeichert werden. Wird dass Programm geladen, wird Speicher in der Größe des BSS-Segments allokiert und mit Nullen initialisiert. Das BSS-Segment wird dabei normalerweise direkt hinter dem DATA-Segment platziert. Diese Behandlung des BSS-Segmentes ist insbesondere für die Nutzung in Sensornetzen von Interesse, da hier bei der Migration von Diensten lediglich die Größe des BSS-Segmentes, nicht jedoch das BSS-Segment selbst übertragen werden muss.

Das Beispiel in Abbildung 5.1 zeigt beispielhaft, wie einzelne Programmteile den verschiedenen Segmenten zugeordnet werden. Die Anweisungen werden in das TEXT-Segment platziert. Die initialisierten Daten werden in das DATA-Segment platziert, wohingegen die unitialisierten Daten dem BSS-Segment zugeordnet werden.

5.3.3 Bindungsprozess

Um ein ausführbares Programm zu erhalten, müssen alle Symbole in der Objektdatei aufgelöst, d.h. durch die entsprechenden Adressen der Variablen und Funktionen ersetzt worden sein. Da ein Programm in der Regel aus mehreren Dateien besteht, sind nicht alle referenzierten Symbole während der Codeerstellung und -übersetzung für eine Datei bekannt. Diese Aufgabe wird vom *Bindelader* (engl.: *linker* oder *binder*) übernommen. Im Folgenden wird der englische Begriff Linker verwendet, der geläufiger ist. Der Linker identifiziert eine Hauptroutine als Einstiegspunkt in das Programm, löst alle symbolischen Referenzen durch Adressen auf und erzeugt aus allen Objektdateien eine einzelne

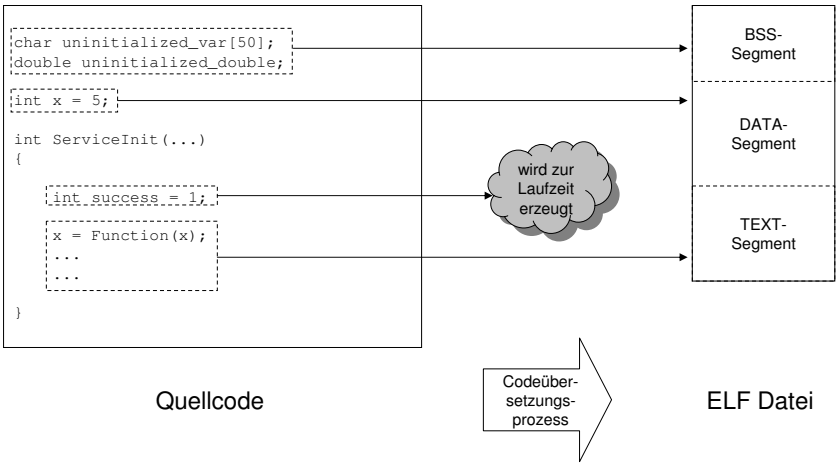


Abbildung 5.1: Zuordnung der einzelnen Programmteile auf die verschiedenen Segmente in einer ELF-Datei

ausführbare Datei. Somit werden vom Linker die TEXT-, DATA- und BSS-Segmente aller Objektdateien in eine einzelne Datei gespeichert.

Durch die Verwendung einer sogenannten *Linker-Description-Datei* können dem Linker Regeln für den Bindungsprozess übergeben werden. Diese Regeln können beinhalten, welche zusätzlichen Programmbibliotheken eingebunden oder wo einzelne Segmente der Objektdateien platziert werden sollen. Zusätzlich können in ELF-Dateien die zuvor genannten Sektionen genutzt werden, um die Inhalte der Segmente weiter zu strukturieren und anzuordnen.

5.4 Architektur

Die Dienste, die in *Surfer OS* zur Laufzeit in eine Sensornetzapplikation integriert werden, stellen unabhängige Maschinencodestücke dar. Wenn der Maschinencode auf einen Knoten migriert, werden die Segmente auf dem Zielgerät neu platziert. Die Adressen, die vom Linker aufgelöst wurden, sind auf dem Zielknoten nicht mehr korrekt. Um den Dienst neu zu platzieren, müssen Relokationsinformationen während des Bindungsprozesses gesammelt werden. Diese Relokationsinformationen werden zusammen mit dem Maschinencode auf den Zielknoten übertragen. Dort werden die Relocationsinformationen und der Maschinencode von einer Schnittstelle auf dem Knoten verarbeitet. Im Falle des *Surfer OS* wird diese Schnittstelle vom *ServiceManager* bereitgestellt. Für jedes Segment des Maschinencodes wird Speicher allokiert. Schließlich wird der Dienst initialisiert und integriert, indem eine vordefinierte Schnittstelle des Dienstes aufgerufen wird.

Um dieses Ziel zu erreichen, müssen die folgenden Schritte ablaufen: Zunächst muss die Architektur der Zielplattform unter dem Gesichtspunkt der Programmierung während der Laufzeit analysiert werden. Als Zweites müssen die Relokationsinformationen identi-

fiziert und bereitgestellt werden, die notwendig sind, um einen Service auf dem Zielknoten fehlerfrei ausführbar zu machen. Abschließend muss der Service eine Schnittstelle bereitstellen, über die der Knoten (hier: der *ServiceManager*) den Service initialisieren und in die laufende Applikation integrieren kann. Im Folgenden wird auf diese Herausforderungen eingegangen.

5.4.1 Zielplattform

Bevor die Migration von Softwaremodulen auf Sensorknoten implementiert werden kann, muss die Funktionsweise der Zielplattform analysiert werden. Abhängig von der Hardwarearchitektur, die dem Sensorknoten zugrundeliegt, wird Code unterschiedlich behandelt und ausgeführt. Diese Vorgänge beispielsweise hinsichtlich der Speicherung von ausführbarem Code haben Einfluss auf die Informationen, die mit dem eigentlichen Servicecode mitgeführt werden müssen. Zudem muss sichergestellt werden, ob die Plattform eine Programmierung des persistenten Flash-Speichers während der Ausführung eines Programms erlaubt oder ob genügend RAM für die Speicherung der Services zur Verfügung steht.

5.4.2 Bereitstellung von Relokationsinformationen

Wie zuvor beschrieben ist ein Programm in verschiedene Segmente wie beispielsweise TEXT, DATA und BSS unterteilt. In der letzten Phase der Codeerstellung werden alle Segmente der Objektdateien vom Linker in eine ausführbare Datei gespeichert. Dabei werden alle übrigen Symbole von Variablen und Funktionen durch Adressen ersetzt.

Wenn der Code neu platziert wird, werden die Segmente an neue Adressen auf dem Zielknoten platziert. Für jedes Segment wurde neuer Speicher auf dem Zielknoten bereitgestellt. Dadurch stimmen die vom Linker während des Codeerstellungsprozesses ersetzten Adressen nicht mehr. Daher ist es notwendig, Relokationsinformationen für jedes Softwaremodul zur Verfügung zu stellen. Diese Relokationsinformationen müssen beinhalten, wo im Programmcode Referenzen in welches Segment gemacht wurden. Anhand dieser Informationen kann der Code auf dem Zielknoten so modifiziert werden, dass an diesen Positionen korrekte Adressen zu den neu zugewiesenen Speicherstellen stehen.

5.4.3 Bereitstellung von Serviceschnittstellen

Nachdem das Modul erfolgreich migriert und die Referenzen innerhalb des Moduls gemäß den neuen Adressen der Segmente korrigiert worden sind, muss der Service in die laufende Applikation eingebunden werden. Wie bereits in Kapitel 4.3.2 beschrieben, müssen für den Integrationsprozess Informationen zwischen der laufenden Applikation und dem empfangenen Servicemodul ausgetauscht werden. Diese beinhalten beispielsweise Informationen über die Funktionalitäten, die der Knoten zu Verfügung stellt, wie Funktionsadressen zu Hardwareschnittstellen. Das migrierte Modul kann Funktionsadressen für

benötigte Funktionalitäten anfordern und dadurch auch Funktionen anderer Services nutzen. Durch diesen Mechanismus können Referenzen innerhalb des Services auf Funktionalitäten des Knotens oder anderer Services während der Laufzeit aufgelöst werden.

Um dies zu realisieren, muss der Knoten eine Schnittstelle anbieten, bei der ein neuer Service entgegengenommen wird. Immer wenn ein Service empfangen wurde, wählt diese Schnittstelle einen vordefinierten Einstiegspunkt in den Service. Durch den Aufruf einer festgelegten Funktion innerhalb des neuen Services wird der Integrationsprozess initiiert. Die Informationen werden mittels der festgelegte Signatur dieser Funktion dem Servicemodul übergeben und das Modul kann gestartet werden. Daher müssen Informationen über diesen Einstiegspunkt dem Knoten zur Verfügung stehen.

5.5 Implementierung

Nachdem in dem vorigen Abschnitt die einzelnen Aufgaben identifiziert wurden, die für die Migration und Integration von Servicemodulen gelöst werden müssen, wird im Folgenden darauf eingegangen, wie die beschriebene Architektur implementiert wurde. Es wird dabei auf die Realisierung auf der *Pacemate* Plattform eingegangen. Es wird beschrieben, wie der Bindungsprozess beim Vorgang der Codeerstellung der Module angepasst wird. Abschließend wird beschrieben, nach welchen Vorgaben der Code für die einzelnen Servicemodule erstellt werden muss.

5.5.1 Migration auf die Pacemate Plattform

Die im Rahmen dieser Arbeit verwendete *Pacemate* Plattform wurde bereits in Kapitel 3 vorgestellt. Für die Migration ist dabei die Eigenschaft von großer Bedeutung, dass der verwendete Mikrocontroller Philips LPC2136 bereits über Programmspeicher verfügt. Der Mikrocontroller kann dabei wie zuvor in Abbildung 3.5 dargestellt den gesamten zur Verfügung stehenden Speicher direkt addressieren. Zusätzliche Techniken wie virtuelle Adressierung oder Speichereinblendung (engl.: *memory mapping*), die größere physikalische Speicheradressen auf Adressen, die von der CPU (engl.: *Central Processing Unit*) angesprochen werden können, abbildet, müssen daher nicht berücksichtigt werden. Um Programme in der Programmiersprache C für den Philips 2136 zu erstellen, wurde der GNU C-Compiler für ARM-Architekturen (*Acorn RISC Machine*) benutzt.

5.5.2 Anpassung des Bindungsprozesses

Wie bereits in Kapitel 5.3 beschrieben, werden im letzten Schritt der Codegenerierung, dem Bindungsprozess, alle symbolischen Referenzen im Maschinencode durch physikalische Adressen ersetzt. Der Linker muss für diesen Vorgang die physikalische Adresse kennen, von der aus der Code ausgeführt wird. Komplexe Betriebssysteme (beispielsweise für Arbeitsplatzrechner), die mehrere Prozesse (Anwendungen) gleichzeitig ausführen,

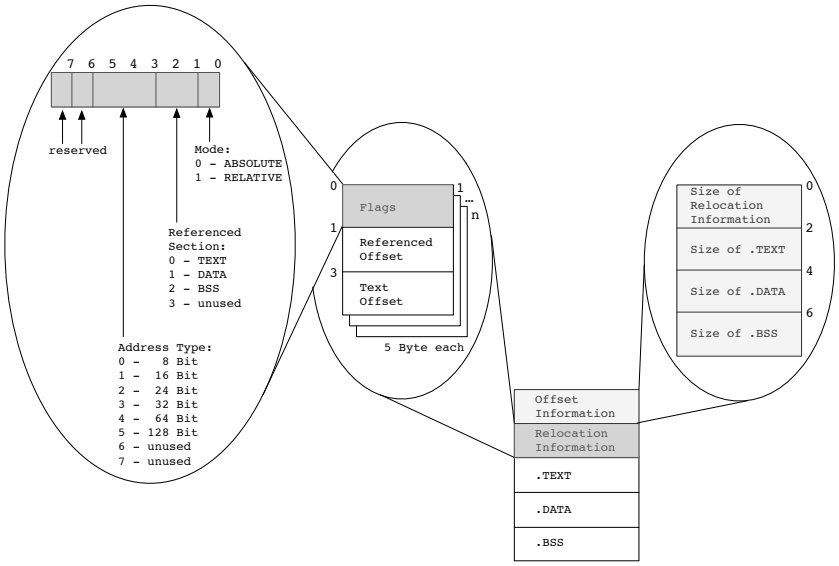


Abbildung 5.2: Dateiformat für ein zu migrierendes Softwaremodul mit aufbereiteten Relokationsinformationen

bieten einen sogenannten virtuellen Speicher, der jedem Prozess das gleiche Speicherlayout anbietet, welches unabhängig vom tatsächlichen physikalischen Speicher ist. Dieser Ansatz erscheint jedoch auf Grund der beschränkten Ressourcen auf Sensorknoten nicht realisierbar.

Um dennoch unterschiedliche Dienste auf einem Sensorknoten ausführen zu können, wird bei dem präsentierten Ansatz der Bindungsprozess angepasst. Dabei wird die Option des *partiellen Bindens* (engl.: *partial linking*), die seitens des GNU Linkers angeboten wird, ausgenutzt. Bei dem partiellen Binden werden Symbole, die an verschiedenen Stellen im physikalischen Speicher platziert werden können, nicht in Adressen aufgelöst. Stattdessen wird eine sogenannte *Relokationstabelle* erstellt, welche eindeutige Symbolnamen auf den Versatz innerhalb eines Segmentes abbildet. Diese Information wird innerhalb der ELF-Datei, die vom Linker erzeugt wird, gespeichert. Wie bereits in Kapitel 5.1 beschrieben, ist der Verwendung des ELF-Formats auf der *Pacemate* Plattform nicht wünschenswert.

Im Rahmen dieser Arbeit wurde eine Methode realisiert, die nur die notwendigen Relokationsinformationen aus der ELF-Datei extrahiert und effizient kodiert. Dazu wurde das Programm `objdump`, welches Teil der GNU Programmierwerkzeuge (*GNU Binutils*) ist, angepasst. `objdump` ermöglicht den Zugriff auf die verschiedensten Informationen in einer Objektdatei. Durch die Anpassung von `objdump` wird eine Ausgabe erzeugt, die jeden Eintrag der Relokationstabelle dabei auf eine Folge von fünf Byte abbildet. Das Format für das zu migrierende Servicemodul ist in Abbildung 5.2 dargestellt.

Das erste der 5 Bytes mit dem Index 0 codiert dabei sogenannte *Flags*, welche die Relokationsinformation genauer spezifizieren. Das Bit 0 innerhalb des ersten Bytes besagt, ob das referenzierte Symbol absolut oder relative addressiert wird. Die Bits 1 bis 2 ge-

ben an, in welchem Segment sich das referenzierte Symbol befindet (TEXT, DATA oder BSS). Diese drei Segmente sind dabei ausreichend, auch wenn innerhalb des Codegenerierungsprozesses mit mehreren Segmenten gearbeitet wird. Mittels einer entsprechenden Linker-Datei kann der gesamte Maschinencode in diese drei Segmente abgebildet werden. Die Bits 3 bis 5 codieren die Adressbreite und somit die Anzahl der Bytes, die ersetzt werden müssen, wenn die physikalische Adresse berechnet wurde. Die Bits 6 bis 7 sind für zukünftigen Gebrauch reserviert.

Die Bytes 1 bis 2 eines 5 Byte Relokationseintrages werden benutzt, um den Versatz des referenzierten Symbols innerhalb des durch das Flag codierten Segments anzugeben. Zwei Bytes decken dabei einen Adressraum von 64 kByte ab, welcher ausreichend für die Segmentgröße eines Servicemodul innerhalb der Sensornetze ist.

Die Bytes 3 bis 4 des Relokationseintrages geben den Versatz der Symbolreferenz im TEXT-Segment an. Dies beschreibt die Stelle im Maschinencode, wo das Symbol referenziert wird und wo die korrigierte Adresse eingefügt werden muss.

Bei der Übertragung eines Servicemoduls werden diese Informationen zusammen mit den Segmenten des Softwaremoduls gespeichert. Zusätzlich ist es notwendig, zunächst die sogenannte Versatzinformationen (engl.: *offset information*) anzugeben. Diese besagen, wie viele Relokationsinformationen in der Datei enthalten sind und wie groß die einzelnen Segmente sind. Diese Informationen werden benötigt, um die Datei erfolgreich zu zerlegen und die neuen Adressen berechnen zu können.

Wenn ein solches Servicemodul empfangen wird, allokiert der Knoten Programmspeicher in der Größe des empfangenen Moduls. Zusätzlich wird im RAM Speicher in der Größe von DATA und BSS auf dem Heap allokiert. Mittels der in dem Modul angegebenen Relokationsinformationen und den Adressen des allokierten Speichers können nun die Referenzen in den Segmenten neu berechnet werden. Das korrigierte Servicemodul wird in den Flash gespeichert und das DATA- und das BSS-Segment werden in den allokierten Speicher im Heap kopiert. Im *Surfer OS* wird diese Aufgabe wie in Kapitel 4.3.2 vom *ServiceManager* übernommen.

5.5.3 Codeerstellung für Servicemodule

Um die Integration eines Dienstes zur Laufzeit zu ermöglichen, wird ein Einstiegspunkt in das Servicemodul benötigt, mittels dessen Informationen zwischen dem Knoten und dem neuen Dienst ausgetauscht werden können. Dieser Einstiegspunkt muss für jeden Dienst identisch sein. Um dies zu erreichen, wurde eine Programmgerüst (engl.: *template*) festgelegt, welches von allen Servicemodulen implementiert wird. Dieses Programmgerüst umfasst zwei Dateien: `service.h` und `service.c`. Die Datei `service.h` legt dabei die verwendeten Datenstrukturen fest. Die eigentliche Schnittstelle zu dem Servicemodul wird in der Datei `service.c` implementiert, die in Auflistung 5.1 angegeben ist.

```
1 #include "service.h"

3 //include your service logic
4 //#include "my_impl.h"
5 //#include ...

7 bool ServiceInit(service_init_in *in, service_init_out *out)
8 {
9     bool success = true;

11    //set the output information
12    //out->info = ...

14    return success;
15 }
```

Quelltext 5.1: Programmgerüst für migrierbares Softwaremodul

Die Funktion `ServiceInit(...)` stellt den Einstiegspunkt zu dem Servicemodul dar. Diese Funktion wird aufgerufen, nachdem der Dienst auf dem Knoten empfangen und relokiert wurde. Die beiden Parameter der Funktion realisieren dabei den Informationsaustausch. Die Datenstruktur des ersten Parameters beinhaltet die Schnittstellen zu den Funktionalitäten des Knotens. Innerhalb des *Surfer OS* sind dies die Schnittstellen zum *ServiceManager* zum Registrieren und Abholen von Funktionalitäten in Form von Funktionsadressen. Der zweite Parameter repräsentiert die Informationen, die vom Dienst an die laufende Applikation übergeben werden. Diese Datenstruktur wird innerhalb der `ServiveInit(...)`-Funktion mit den entsprechenden Daten befüllt. Auf diese Weise lassen sich Informationen zwischen dem neuen Dienst und dem laufenden System (beispielsweise *Surfer OS*) austauschen. Die verwendeten Datenstrukturen müssen dabei sowohl seitens der Applikation als auch seitens des Dienstes identisch definiert sein.

Nachdem die Funktion zur Initialisierung festgelegt worden ist, muss diese an eine vorbestimmte Position innerhalb des Servicemoduls platziert werden, so dass sie seitens der Applikation gefunden und ausgeführt werden kann. Die Option `-fdata-sections -ffunction-sections` bei der Codeerstellung des Dienstes erlaubt die bereits aufgeführte Strukturierung der Segmente in Sektionen. Diese Option generiert eine eigene Sektion innerhalb der betreffenden Segmente für jede Funktion und jede Variable. Dadurch lassen sich die einzelnen Sektionen im Bindungsprozess referenzieren. Mittels einer speziellen Linker-Descrition-Datei, die in Auflistung 5.2 aufgeführt ist, wird in den Zeilen 6 bis 9 erwirkt, dass die Funktion `ServiceInit(...)` am Anfang des Maschinencodes an Adresse 0x0000 steht. Zudem platziert der Linker nach den Angaben in der Linker-Descrition-Datei die Segmente in der Reihenfolge TEXT, DATA und BSS hintereinander.

```
1  /* Section Definitions */
2  SECTIONS
3  {
4    /* first section is .text which is used for code */
5    .text 0x00000:
6    {
7      /* function code from .service-init-section */
8      service.o (.text.ServiceInit)
9
10     /* remaining code and string constants*/
11   *(.text .text.* .rodata .rodata.*)
12   }
13   .data :
14   {
15    /* all data sections */
16    *(.data .data.* )
17   }
18   .bss :
19   {
20    /* all bss sections */
21    *(.bss .bss.*)
22   }
23 }
```

Quelltext 5.2: Linker-Descrition-Datei für migrierbare Softwaremodule

Das Modul ist zu diesem Zeitpunkt bereits relokiert und in den Flash Speicher geschrieben. Die laufende Applikation wandelt die Adresse des Anfangs des TEXT-Segmentes in eine Funktionsadresse mit der festgelegten Signatur der `ServiceInit(...)`-Funktion um und ruft diese auf. Nachdem die Funktion mit den entsprechenden Parametern ausgeführt wurde, ist der Integrationsprozess abgeschlossen.

5.5.4 Erweiterungen für eine Verwendung in Surfer OS

Die präsentierte Verfahrensweise beschreibt einen generellen Ansatz zur Lösung des Relokationsproblems sowie der Integration von Softwaremodulen. Im speziellen Anwendungsfall des *Surfer OS* wird der Ansatz noch erweitert. Zum einen übernimmt der *ServiceManager* die Aufgaben des Zerlegens in die Segmente, des Relokierens und des Speicherns des Servicemoduls. Zum anderen entscheidet der *ServiceManager*, ob Dienste eingebunden oder ersetzt werden. Die Informationen, die zwischen *ServiceManager* und neuem Dienst ausgetauscht werden, enthalten neben den Schnittstellen zum *ServiceManager* ebenfalls Informationen wie Serviceidentifikation und Servicenamen, die für die Koordinierung der Dienste von Nöten sind und vom *ServiceManager* ausgewertet werden.

Das Programmgerüst wird um weitere Funktionen neben der `ServiceInit(...)` erweitert. So erhält jeder Service eine `ServiceStart()`-, `ServiceStop()`- und eine `ServiceCleanUp()`-Funktion, die alle automatisch beim *ServiceManager* registriert werden. Diese Funktionen dienen der höheren Flexibilität bei der Nutzung von Diensten innerhalb von *Surfer OS*. So ist der Start eines Dienstes durch die separate Funktion `ServiceStart()` von der Initialisierung getrennt. Im Fehlerfall bei der Registrierung beispielsweise mangels zur Verfügung stehenden Speichers kann die Initialisierung so leicht wieder rückgängig gemacht werden. Nachdem die Initialisierung erfolgreich abgeschlossen wurde, wird seitens des *ServiceManagers* der Service durch den Aufruf der Funktion `ServiceStart()` gestartet. Zudem wird in Kombination mit der Funktion `ServiceStop()` ermöglicht, dass ein Dienst unterbrochen werden kann, ohne dass er neu initialisiert werden muss. Die Funktion `ServiceCleanUp()` wird beim Entfernen eines Dienstes automatisch seitens des *ServiceManagers* gerufen. In dieser Funktion muss der Programmierer den dynamisch belegten Speicher wieder freigeben und die Rückruffunktionen für Ereignisse deregistrieren. Somit wird ein sauberes Entfernen der Dienste realisiert.

5.6 Servicebindung und zustandsbehaftete Migration

Die Migration von Diensten ermöglicht nicht nur die Komposition von Applikationen auf einzelnen Knoten. Vielmehr wird durch das hier realisierte Migrationsverfahren die Bindung zwischen Knoten und Dienst derart gelockert, dass ein Dienst eine zuvor eingegangene Bindung an einen Knoten aufgeben und auf einen anderen Knoten migrieren kann. Dies ermöglicht die Umsetzung neuer Strategien bei der Applikationserstellung. Einzelne Dienste können förmlich durch das Netz wandern und zum Beispiel Daten einsammeln oder gar Konfigurationen auf den Knoten vornehmen.

Dabei kann durch das realisierte Verfahren sogar der Zustand des Dienstes bei der Migration erhalten bleiben. Das Orginal des Servicemoduls bestehend aus Versatzinformationen, Relokationsinformationen und den einzelnen Segmenten ist persistent im Flash Speicher (ROM) gehalten, wie in Abbildung 5.3 dargestellt. Gleichzeitig ist jeweils eine Kopie des DATA- und des BSS-Segments im RAM vorhanden, welche die aktuellen Werte der Variablen hält. Migriert der Service zustandslos, so werden die Orginaldaten aus dem ROM zur Migration verwendet (wie in Abbildung 5.3(a)). Werden bei der Migration nun anstatt der orginalen DATA- und BSS-Segmente die aktiven Kopien im RAM verwendet wie in Abbildung 5.3(b), wird der Zustand der aktuellen Variablen auf den neuen Knoten übertragen. So kann ein Service auf einem anderen Knoten weiterarbeiten.

Neben der Migration ist dabei auch eine Replikation der Dienste leicht realisierbar. Das im Flash Speicher befindliche Servicemodul mit seinen Relokationsinformationen wird über Funk an einen anderen Knoten gesendet. Im Gegensatz zur Migration verbleibt jedoch der Service auf dem Ursprungsknoten und wird nicht entfernt.

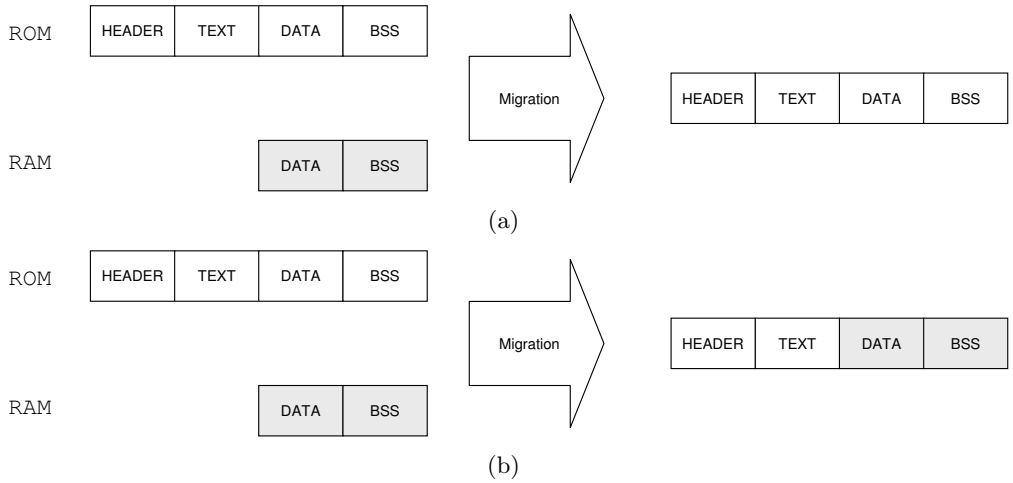


Abbildung 5.3: Migration eines Dienstes von einem Knoten (a) zustandslos, (b) zustandsbehaftet

5.7 Evaluation

Die Multi-hop Migration von Programmcode wurde im Rahmen dieser Arbeit innerhalb des *Surfer OS* realisiert. Weiterhin wurden etliche Servicemodule geschrieben, die auf die *Pacemate*-Sensorknoten migrieren und eine laufende *Surfer OS*-Applikation bilden. In diesem Abschnitt wird der realisierte Ansatz evaluiert. Es wird dabei die zusätzlich benötigte Menge an Relokationsinformationen für einzelne Dienste untersucht. Außerdem wird die benötigte Rechenlast auf den Knoten betrachtet, die bei der Migration entsteht. Abschließend werden die durch die zugrundeliegende *Pacemate*-Plattform bedingten Effekte beim Zugriff auf dem Flashspeicher untersucht.

Der realisierte Ansatz erfordert, dass mit jedem migrierbaren Servicemodul Relokationsinformationen bereitgestellt werden. Tabelle 5.1 zeigt die Größe und den Anteil der Relokationsinformationen für verschiedene Dienste. Bei der absoluten Größe der Dienste muss berücksichtigt werden, dass diese die durch das *Surfer OS* gegebenen Erweiterungen zur Servicekoordination beinhalten. Die benötigte Menge an Relokationsinformationen liegt im Durchschnitt bei 15,3 % der gesamten zu übertragenden Servicemodulgröße. Anhand der in Tabelle 5.1 präsentierten Ergebnisse ist zu beobachten, dass bei den kleineren Diensten der Anteil der Relokationsinformationen tendenziell höher ist. Dies ist darin begründet, dass das Verhältnis von Funktionsaufrufen zu implementierter Logik größer ist und somit im Vergleich zu der Größe des Gesamtcode mehr Referenzen ersetzt werden müssen.

Bei der Realisierung der Migration stand im Vordergrund die Logik, die auf dem Knoten vorhanden sein muss, sowie die durchzuführenden Berechnungen zu reduzieren. So muss auf den Sensorknoten weder Programmcode interpretiert werden, wie beim Ansatz der virtuellen Maschinen, noch muss ein Programmmodul auf der Maschine gebunden wer-

Softwaremodules	Headergröße (in byte)	Gesamtgröße (in byte)	Overhead
blinking led	136	760	17,9 %
beacon service	136	784	17,3 %
neighbourhood monitor	248	1496	16,6 %
RAM Monitor	264	1528	17,3 %
Fluten	340	2632	12,9 %
Routing Radio Stack	428	2936	14,6 %
DySSCo	644	5572	11,6 %
Grape	840	6076	13,8 %
Durchschnittlicher Overhead			15,3 %

Tabelle 5.1: Größen für verschiedene Softwaremodule und Relokationsinformationen

den. In der hier präsentierten Lösung müssen lediglich Adressen in dem Maschinencode ersetzt werden. Durch die in der Entwicklungsumgebung bereitgestellten Relokationsinformationen reduziert sich die Berechnung einer neuen Adresse zur Laufzeit auf eine einfache Addition.

Um den auf der *Pacemate*-Plattform zur Verfügung stehenden Speicher effizient zu nutzen, werden die empfangenen Dienste in den 256 kbyte großen Flash Speicher geschrieben. Der Zugriff auf den Flash-Speicher benötigt auf der *Pacemate*-Plattform 1 ms pro 256 byte. Anhand der in Tabelle 5.1 angegeben Größen für einige Beispieldienste lässt sich erkennen, dass das Speichern eines Dienstes in Abhängigkeit von der Größe mehrere Millisekunden in Anspruch nehmen kann.

Wenn ein Servicemodul im *Surfer OS* empfangen und in den Flash-Speicher geschrieben wird, müssen die Inhalte der in Kapitel 4.4 beschriebenen Datenstrukturen zur Speicherverwaltung angepasst werden. Zudem kann der Empfang eines Servicemoduls erfordern, dass andere Module, beispielsweise ältere Versionen dieses Services, entfernt werden müssen, um belegten Speicher wieder freizugeben. Daten, die bereits in den Flash geschrieben worden sind, können jedoch nicht mit neuen Werten überschrieben werden. Der Flash-Speicher der *Pacemate*-Plattform kann nur sektorenweise gelöscht und neu beschrieben werden. Wie in Kapitel 3.3 beschrieben, ist der Speicher des LPC2136 Mikrocontrollers in verschiedene Sektoren der Größen 4 und 32 kbyte unterteilt. Änderungen am Inhalt des Flash-Speichers haben somit zur Folge, dass der komplette betroffene Sektor und somit bis zu 32 kbyte ersetzt werden müssen. Der betroffene Sektor muss somit zunächst in einen anderen Sektor der gleichen Größe kopiert werden. Anschließend muss der Sektor gelöscht werden, bevor die Daten modifiziert und zurückgeschrieben werden. Dies bedeutet, dass Änderungen im Flash-Speicher mit hohem Rechen- und Zeitaufwand verbunden sind. Tabelle 5.2 führt dabei die auf der *Pacemate*-Plattform benötigte

Zeit auf, um Änderungen an einem Sektor A der Größe 32 kbyte unter Verwendung des temporären Sektors B vorzunehmen. Man sieht, dass das Hinzufügen eines Dienstes im schlechtesten Fall über eine Sekunde in Anspruch nimmt.

Operation	Zeit
schreibe Inhalt A nach B	$\frac{32\ \text{kbyte}*1024}{256\ \text{byte}} * 1\ \text{ms} = 128\ \text{ms}$
lösche A	400 ms
schreibe Inhalt B nach A	128 ms
lösche B	400 ms
Gesamtzeit	1056 ms

Tabelle 5.2: Benötigte Zeit, um den Inhalt eines 32 kbyte Flashspeichersektors A zu modifizieren

5.8 Ergebnis

In diesem Kapitel wurden zunächst existierende technische Realisierungen zur Migration und Integration einzelner Softwaremodule betrachtet. Dabei wurden die Arbeiten auf die Anwendbarkeit innerhalb *Surfer OS* geprüft. Hierbei wurde insbesondere die Anforderung geprüft, ob auch vom System bereits angebotene Dienste ersetzt werden können. Zudem wurden die Möglichkeiten zur Portierung auf verschiedene Hardwareplattformen untersucht.

Basierend auf den formulierten Anforderungen wurden die Grundlagen der Codeerstellung untersucht. Auf dieser Basis ist ein Migrationsmechanismus entwickelt worden, der auf verschiedene Hard- und Softwareplattformen übertragbar ist. Das präsentierte Verfahren führt dabei den Großteil der notwendigen Berechnungen in der Entwicklungsumgebung aus. Dies wird durch die Modifikation des Bindungsprozesses bei der Codeerstellung und die damit erzeugte Bereitstellung von Relokationsinformationen erreicht. Dabei werden die auf dem Sensorknoten auszuführenden Berechnungen auf das Zerlegen der empfangenen Daten und eine einfache Addition je Referenz reduziert. Das Verfahren beinhaltet zudem eine Bereitstellung einer Schnittstelle für die Servicemodule, die eine Integration des Moduls in eine laufende Applikation ermöglicht. Diese Schnittstelle wurde für die Verwendung in *Surfer OS* erweitert, um ein einheitliches Programmgerüst festzulegen, welches eine saubere Integration, Ausführung, Beendigung und Entfernung von Servicemodulen ermöglicht. Ein Programmierer kann mittels dieses Programmgerüstes leicht neue Dienste für eine Applikation erstellen. Die Komplexität des *Surfer OS* bleibt dabei dem Programmierer verborgen.

Mittels des präsentierten Verfahrens ist zudem die Möglichkeit zur statusbehafteten Migration von Diensten gegeben. Ein Dienst kann die Bindung an einen Sensorknoten auf-

geben und unter Beibehaltung seines aktuellen Status auf einen anderen Knoten migrieren. Dadurch ist die Grundlage für verschiedene selbstorganisierende Verfahren gelegt, die Dienste basierend auf im Netzwerk gemessenen Metriken migrieren und platzieren.

Die technische Realisierung der Migration stellt dabei einen ersten Schritt dar. Weiterführende Konzepte, wie insbesondere Transaktionen, realisieren dabei aufbauend auf der Migration konsistente und robuste Abläufe insbesondere bei auftretenden Fehlerfällen, wie beispielsweise Übertragungsfehlern. Erste Ansätze zur Übertragung dieser Konzepte in Sensornetze wurde in [89] unter Mitarbeit des Autors veröffentlicht.

6 DySSCo – Adaptive Dienstverteilung

In den beiden vorangegangenen Kapiteln hat der Autor das von ihm entwickelte Sensorknotenbetriebssystem *Surfer OS* vorgestellt, das den Grundbestandteil der in dieser Arbeit realisierten service-orientierten Infrastruktur bildet. Außerdem ist ein Mechanismus zur Migration und Replikation von Diensten realisiert worden. Diese beiden Komponenten bilden die geforderte Grundlage für flexibel anpassbare Sensornetzanwendungen und bieten dem Anwendungsprogrammierer das Paradigma der Service-Orientierung als Abstraktion der verteilten Anwendung. Der Benutzer kann Dienste mittels *Surfer OS* auf einzelne Knoten oder im gesamten Netz verteilen, die sich dadurch zu einer Sensornetzanwendung zusammensetzen. Gegebenenfalls kann der Benutzer somit auf sich ändernde Anforderungen oder geänderte Netztopologien reagieren, indem er weitere Dienste in das Netz einspielt oder bestehende Dienste entfernt oder ersetzt. Diese Anpassungen bedürfen jedoch der menschlichen Interaktion mit dem Netz. Um Entscheidungen zur Migration zu treffen, müssen dem Benutzer zudem Daten über den Netzstatus wie beispielsweise Positionen einzelner Knoten oder Nachbarschaftsgrößen in einzelnen Bereichen des Netzes zur Verfügung stehen. Diese muss er zudem richtig interpretieren können, um zu entscheiden, welcher Dienst auf welche Knoten migriert werden muss.

Neben der bisher realisierten Möglichkeit, das Netz flexibel anzupassen, muss nun eine weitere Abstraktion realisiert werden, die es dem Benutzer ermöglicht, Anforderungen für seine Anwendung zu formulieren, die vom Sensornetz eigenständig erfüllt werden. Eine solche Abstraktion erhöht die Transparenz des Sensornetzes als verteiltes System und verbirgt die zugrundeliegende Komplexität. Zudem macht es eine ständige Wartung im Sinne der manuellen Verteilung von Diensten im Bedarfsfall überflüssig, da das Sensornetz eigenständig auf Veränderungen reagieren kann. Es wird zudem die Stabilität der Anwendung in der Art erhöht, dass das Sensornetz auch auf zwischenzeitliche Veränderungen zeitnah reagieren kann. Somit kann das Sensornetz den Ablauf der Anwendung im Rahmen der seitens des Benutzers formulierten Anforderungen zusichern. Aus dieser Motivation heraus hat der Autor den adaptiven Dienstverteilungsalgorithmus *DySSCo* (*Dynamic Self-organizing Service Coverage*) [64][66] entwickelt, der es dem Anwender ermöglicht, seine Anwendung in Form von prozentualen Abdeckungen einzelner Dienste im Netz zu formulieren.

In diesem Kapitel werden existierende Arbeiten zu dem Problem der Serviceverteilung vorgestellt. Anschließend wird der algorithmische Ansatz des *DySSCo*-Algorithmus vorgestellt. Der Algorithmus wird graphentheoretisch analysiert und auf sein Konvergenzverhalten untersucht. Darauffolgend wird auf die Implementation des Algorithmus eingegangen. Der Algorithmus wird abschließend evaluiert, und die Ergebnisse werden zusammengefasst.

6.1 Verwandte Arbeiten

Im Folgenden werden verschiedene Algorithmen vorgestellt, die das Problem der Dienstverteilung in Netzen behandeln. Die meisten Ansätze legen dabei das *Facility Location Problem* zugrunde. Bei diesem Problem wird versucht, aus einer Menge von Standorten (engl.: *location*) eine Teilmenge von Standorten für Einrichtungen (engl.: *facilities*) so auszuwählen, dass eine gegebene Metrik minimiert wird, welche die Entfernungen zu den sogenannten *clients* und die Kosten der Eröffnung einer *facility* quantifiziert. Dabei wird ein *client* einer *facility* zugeordnet. Es wird zwischen dem *capacitated Facility Location Problem* und dem *uncapacitated Facility Location Problem* unterschieden. Im Falle des *capacitated Facility Location Problem* ist die Zahl der *clients* pro *facility* begrenzt. Ein Spezialfall des *Facility Location Problems* stellt das *k-Median Problem* dar, bei dem die Kosten für die Eröffnung einer *facility* vernachlässigt werden, jedoch die Anzahl der *facilities* fest gewählt ist.

Ein anschauliches Beispiel für das *Facility Location Problem* lässt sich in der Logistik finden. Dabei sollen für viele Marktfilialen eine feste Anzahl an Versorgungszentren platziert und gebaut werden, so dass die Entfernungen zu allen Filialen möglichst gering ist. In dem hier aufgeführten Fall findet dieses Optimierungsproblem seine Anwendung in der Verteilung von Diensten auf ausgewählten Knoten im Netz.

6.1.1 Zentralisierter Ansatz

Furuta et al. [24] stellen in ihrer Arbeit einen zentralisierten Algorithmus vor, der ein Netzwerk in Gruppen (engl.: *cluster*) aufteilt und für jede Gruppe einen Knoten als sogenannten *clusterhead* bestimmt. Der *clusterhead* erfüllt dabei spezielle Aufgaben für die Gruppe und bietet damit im übertragen Sinne Dienste für die Gruppe an.

Der präsentierte Ansatz basiert dabei den *LEACH-C* Algorithmus (*Low Energy Adaptive Clustering Hierarchy Centralized*) [31]. Dabei übernimmt eine Basisstation die Kontrolle über die Gruppierung innerhalb des Netzes. Das Auswahlverfahren ist dabei in mehrere Runden unterteilt, wobei jede Runde wiederum in eine sogenannte *clustering phase* und *data transmission phase* unterteilt ist. In der *clustering phase* sendet jeder Knoten Informationen über seine Position und seinen Batteriestatus an eine Basisstation. Basierend auf diesen Informationen wird in *LEACH-C* das *uncapacitated k-Median Problem* gelöst, mittels dessen die *clusterheads* bestimmt werden. In der *data transmission phase* senden die nicht als *clusterhead* gewählten Knoten ihre Daten an den ihnen zugewiesenen *clusterhead*. In einer Erweiterung wird auf der Basisstation basierend auf den zur Verfügung stehenden Informationen das *uncapacitated Facility Location Problem* gelöst, welches eine höhere Flexibilität in der Anzahl der *clusterheads* ermöglicht.

In der Arbeit konnte anhand von Simulationen gezeigt werden, dass durch diesen Ansatz die Auswahl dahingehend verbessert wird, dass die Funktionalität des Netzes über einen längeren Zeitraum erhalten bleibt. Die näherungsweise optimale Platzierung der Dienste

bewirkt ein geringeres Nachrichtenaufkommen, welches die Lebensdauer der Knoten und somit die Funktionalität des Netzes verlängert.

6.1.2 Mehrheits-Votieren

Um mehrere Dienste in einem Netz zu replizieren und zu verteilen, stellen Krivitski et al. [49] in ihrer Arbeit einen verteilten Ansatz vor, um das *uncapacitated Facility Location Problem* zu lösen. Um dieses Optimierungsproblem im Netz zu lösen, wird mittels des sogenannten *Bergsteigeralgorithmus* (engl.: *hill climbing*) das Optimum über mehrere Runden angenähert. Dabei stimmen die Knoten lokal über den nächsten Schritt des Bergsteigeralgorithmus ab. In simulativen Untersuchungen konnte gezeigt werden, dass das Verfahren gegen ein globales Optimum für die Platzierung von Diensten konvergiert und in Hinblick auf die notwendigen Kontrollnachrichten mit der Größe des Netzes skaliert.

6.1.3 Iteratives Verhandeln

Frank et al. [22] präsentieren in ihrer Arbeit einen Algorithmus, in dem potientielle vorherbestimmte *clients* mit potentiellen *facilities* verhandeln. Dabei wird basierend auf lokal ausgetauschten Topologieinformationen wie Linkqualitäten zwischen den Knoten ausgehandelt, welche Knoten eine *facility* eröffnen. In der Arbeit wird dabei nachgewiesen, dass der verteilte Algorithmus äquivalent zu einem zentralisierten Algorithmus ist. Um sich der in Sensornetzen vorhandenen Dynamik anzupassen, prüft ein *client*, ob seine gewählte *facility* noch präsent und optimal ist, und wählt diese gegebenfalls neu aus. Als weitere Option ist es möglich, den gesamten Auswahlprozess neu zu starten. Der Algorithmus wurde sowohl simulativ als auch in einem Experiment mit 13 Sensorknoten evaluiert.

6.1.4 Verteilte lineare Programmierung

Moscibroda et al. [74] nutzen ebenfalls ein iteratives Verfahren, um das *uncapacitated Facility Location Problem* verteilt zu lösen. Das Verfahren realisiert dabei eine verteilte *primal-dual-Variante* [3] zur Approximierung der Lösung eines *linearen Programms* (auch *lineare Optimierung*). Die Autoren zeigen dabei, dass die Annäherung an die optimale Lösung zur Platzierung der Dienste durch die Anzahl der verwendeten Nachrichten abgeschätzt werden kann und durch diese begrenzt ist. Das vorgestellte Verfahren wurde mathematisch analysiert und bewiesen, jedoch nicht simulativ oder experimentell untersucht.

6.1.5 Mobilitätsbasiertes Verfahren

Li et al. [58] stellen in ihrer Arbeit einen Ansatz vor, der die ständige Verfügbarkeit von Diensten auch bei der Partitionierung des Netzes in mobilen Szenarien sicherstellt.

Basierend auf den Bewegungsrichtungen und Geschwindigkeiten der Knoten trifft der Algorithmus eine Vorhersage, wann ein Service für eine Gruppe von Knoten nicht mehr verfügbar ist. Dabei muss der Service seine Nutzer und sich selber verschiedenen Mobilitätsgruppen zuordnen. Die Informationen über die Bewegungsrichtungen und Geschwindigkeiten werden von den Nutzern zusammen mit den Serviceanfragen gesendet (engl.: *piggybacking*), so dass der Service die zur Zuordnung notwendigen Informationen erhält. Erkennt der Service, dass sich die Gruppe seiner Nutzer partitionieren wird, repliziert er sich auf einen Knoten der anderen Mobilitätsgruppe. In der Arbeit wird simulativ gezeigt, dass auf diese Weise die Verfügbarkeit des Services für jeden Nutzer aufrechterhalten werden kann.

6.1.6 Lokaler Ansatz

Laoutaris et al. [52] adressieren in ihrer Arbeit das Problem der Skalierbarkeit bei der Lösung des *uncapacitated Facility Location Problems* und des *uncapacitated k-Median Problems*. Dabei werden initial verschiedene *facilities* (Instanzen eines Services) im Netz verteilt. Diese wandern dann an die optimalen Positionen im Netz und replizieren oder löschen sich gegebenenfalls. Die Entscheidung für die Migration und Replikation wird dabei je nach Problemstellung durch das *uncapacitated Facility Location Problem* oder das *uncapacitated k-Median Problem* gelöst. Um die Skalierbarkeit zu erreichen, wird dabei nicht die gesamte Netzwerktopologie berücksichtigt, sondern das Problem wird nur in einer zuvor festgelegten n-hop-Nachbarschaft gelöst. Den platzierten *facilities* müssen dabei exakte Informationen über die Topologie dieser Nachbarschaft vorliegen. Die *facility* berücksichtigt dabei Serviceanfragen, die von außerhalb des festgelegten Bereichs kommen, indem diese auf die äußeren Knoten der n-hop-Nachbarschaft abgebildet werden. Der Ansatz wurde simulativ in künstlichen Topologien und Internettopologien für n-hop-Nachbarschaften mit n=1 und n=2 evaluiert.

6.1.7 Abdeckungsbasierter Ansatz

Sailhan et al. [91] behandeln das Problem der Platzierung von Dienstverzeichnissen in mobilen Netzen. Dabei soll kein zentrales Verzeichnis existieren, sondern verschiedene verteilte Instanzen des Verzeichnisdienstes. Dabei senden Knoten, die den Verzeichnisdienst anbieten, in festen Intervallen Nachrichten, sogenannte *advertisement beacons*. Erhält ein Knoten über einen bestimmten Zeitraum keine solche Nachricht, startet er einen Auswahlprozess, in dem er in einer festgelegten n-hop-Nachbarschaft eine sogenannten *election Nachricht* versendet. Jeder Knoten antwortet dabei an den Absender, ob er die Aufgabe für einen Verzeichnisdienst übernehmen möchte, basierend auf den ihm zur Verfügung stehenden Ressourcen. Diese Antwort beinhaltet dabei Informationen über die Nachbarschaftsgrößen sowie die Anzahl der Verzeichnisdienste in der n-hop-Nachbarschaft des Knotens. Der Knoten, der den Auswahlprozess gestartet hat, wählt nun basierend auf der Abdeckung der Verzeichnisdienste den Knoten mit der geringsten

Abdeckung. Die Funktionsweise des Algorithmus wurde mittels des Netzwerksimulators *ns2* untersucht und evaluiert.

6.2 Anforderung

Um die durch das *Surfer OS* und die Migration geschaffene Grundlage für eine Programmierabstraktion zu nutzen, muss ein Ansatz zur selbstorganisierenden Dienstverteilung es dem Anwender ermöglichen, seine Anforderungen abstrakt zu formulieren. Weiterhin steht im Fokus dieser Arbeit, dass die vorgestellten Paradigmen und Algorithmen nicht nur theoretisch funktionieren, sondern auch praktisch einsetzbar sind. Um dies zu erreichen, müssen die dem Algorithmus zugrundeliegenden Annahmen richtig gewählt sein. Die Funktion des Algorithmus darf durch die Dynamik des Netzes und möglichen Nachrichtenverlust nicht beeinflusst werden, da gerade dies eine Eigenschaft des Netzes ist, die vor dem Nutzer verborgen werden soll. Zudem darf die Funktion des Verteilungsalgorithmus nicht abhängig von der Funktion anderer Algorithmen sein, um die Funktionalität gewährleisten zu können.

Ein zentralisierter Ansatz wie in [24] erscheint dabei nicht anwendbar, da dieser für große Netze nicht skaliert. Informationen über das gesamte Netzwerk müssen gesammelt und an einer zentralen Stelle (einer Basisstation) ausgewertet werden. Der dabei entstehende Kommunikationsaufwand ist zu groß und belastet zudem insbesondere die Knoten in der Nähe der Basisstation, da diese vermehrt Nachrichten weiterleiten müssen. Zudem bedarf es für diesen Ansatz wie auch für andere Verfahren, die Topologieinformationen auf speziellen Knoten benötigen (beispielsweise [52]), eines robusten Wegewahlverfahrens.[1]

Eine weitere Annahme in [24], der nicht immer Genüge getan werden kann, ist die Abhängigkeit von Positionsinformationen oder wie in [58] von Geschwindigkeits- und Richtungsinformationen sowie speziellen Bewegungsmustern. Sofern die Knoten im Netz nicht über GPS-Empfänger (*Global Positioning System*) verfügen, sind diese Informationen nur über weitere komplizierte Algorithmen zu erhalten, die im Rahmen dieser Arbeit nicht behandelt werden. Die vorgestellten Verfahren sind abhängig von der Verfügbarkeit der Positionsinformationen.

Verfahren, die iterativ über mehrere Runden die Position und Verteilung eines Dienstes berechnen wie [49][22], setzen voraus, dass die Topologie sich innerhalb dieser Optimierungsphase nicht verändert. Ob die Konvergenz bereits bei geringen Topologieänderungen sichergestellt werden kann, ist fraglich.

Die präsentierten verwandten Arbeiten, deren Hauptaugenmerk auf der optimalen Verteilung von Diensten unter einer gegebenen Metrik basieren, sind mit der Ausnahme von [22] lediglich theoretisch und simulativ untersucht worden. Der Beweis für die Praxistauglichkeit der präsentierten Algorithmen steht in den meisten Fällen also noch aus. Zudem scheint das Ziel, eine optimale Position für einen Dienst im Sinne einer Kostenmetrik gegenüber den Nutzern zu finden, unter der Annahme von sich ändernden

[1] Auf das Problem der Wegewahl in Netzen wird in Kapitel 7 eingegangen.

Topologien und gegebenfalls Mobilität schwierig. Es ist leicht ersichtlich, dass sich in diesem Falle die optimalen Positionen mit der Dynamik des Netzes verändern. Das Optimieren würde dabei unter Umständen höhere Kommunikationskosten verursachen als die nicht optimale Position des Dienstes.

6.3 Algorithmischer Ansatz

Im Gegensatz zu dem von den verwandten Arbeiten verfolgten Ziel, die optimale Anzahl und Platzierung von Instanzen eines Dienstes in Abhängigkeit von den Dienstnutzern (Knoten im Netzwerk) zu bestimmen, legt der hier präsentierte *DySSCo*-Algorithmus zu Grunde, dass der Applikationsentwickler eine prozentuale Abdeckung für einzelne Dienste im Netz formuliert. Diese geforderte Abdeckung für einen Dienst s wird als $c_{demanded}(s)$ definiert. Der *DySSCo*-Algorithmus erhält diese geforderte Abdeckung für jeden Dienst innerhalb des Netzes aufrecht. Dabei nutzt der Algorithmus lediglich lokale Informationen und ist nicht abhängig von Algorithmen, wie beispielsweise für Wegewahl oder Positionsbestimmung.

Der Algorithmus kann genutzt werden, um einen Dienst auf die entsprechenden Knoten zu migrieren oder vorhandene Dienste auf den Knoten zu aktivieren. Im Folgenden wird dabei das Aktivieren eines Dienstes als Synonym für beide Fälle verwendet. Der Status eines Dienstes wird somit als aktiviert oder deaktiviert beschrieben.

Die Grundidee ist, dass durch die lokale Aufrechterhaltung der Abdeckung die globale Abdeckung für einen Service im ganzen Netz erreicht wird. Informationen über die aktivierten Dienste auf einem Knoten n werden von diesem Knoten in einem festgelegten Intervall ausgesandt. Diese Information wird dabei jedoch nicht weitergeleitet, sondern nur von den direkten Nachbarn ausgewertet. Die Information verbreitet sich dabei dennoch automatisch durch das gesamte Netzwerk, ohne abhängig von einem Wegewahlverfahren zu sein.

Ein Knoten n kann anhand der Information über seine Nachbarschaftsgröße und des Status eines Dienstes auf den benachbarten Knoten eine aktuelle lokale Abdeckung berechnen. Basierend auf dieser Information kann der Knoten n über den lokalen Status für den Dienst s entscheiden.

Der Algorithmus reagiert dabei auf Netzwerkdynamik sowie auf Knoten, die das Netzwerk verlassen (beispielsweise wegen leeren Batterien), oder neu hinzukommende Knoten. Zudem unterstützt der Algorithmus, dass Anforderungen an die Abdeckung eines Dienstes seitens des Benutzers im Laufe der Applikation nachträglich angepasst werden.

6.4 Implementierung

Der *DySSCo*-Algorithmus nutzt lokale Informationen auf jedem Knoten, um eine vorgegebene Abdeckung aufrecht zu erhalten. Dabei trifft jeder Knoten autark eine eigene Entscheidung, ob ein Dienst aktiviert oder deaktiviert wird. Im Folgenden wird

s_id	$c_{demanded}(s)$ (in %)
5	50
12	33
17	75
...	...

Tabelle 6.1: Beispiel einer *DySSCo*-Nachricht mit Serviceidentifikation und geforderter Abdeckung

auf das Grundverhalten, den Entscheidungsprozess sowie die Parameter des *DySSCo*-Algorithmus eingegangen.

6.4.1 Grundverhalten

Die geforderte Abdeckung für einen Service s wird wie bereits eingeführt mit der Größe $c_{demanded}(s)$ beschrieben. Um dies zu erreichen und etwaige Veränderungen in der Netztopologie zu erkennen, sendet jeder Knoten in einem Zeitintervall t eine Nachricht (engl.: *beacon*), die in der direkten 1-hop-Nachbarschaft gehört wird. Diese Nachricht enthält die global eindeutigen Serviceidentifikationen s_{id} der aktuell auf dem Knoten aktivierten Dienste s sowie deren geforderte Abdeckungen $c_{demanded}(s)$. Tabelle 6.1 zeigt ein Beispiel für eine solche Nachricht. Hat ein Knoten keine aktivierten Dienste, sendet dieser dennoch eine leere Nachricht, wodurch die benachbarten Knoten Informationen über die aktuelle Nachbarschaftsgröße bekommen. Anhand der Nachricht erhält ein Knoten Informationen über den Status der Dienste in der Nachbarschaft. Basierend auf diesen Informationen sowie der Nachbarschaftsgröße, kann jeder Knoten berechnen, ob die berechnete Abdeckung für jeden Service lokal in der direkten Nachbarschaft größer als die geforderte globale Abdeckung $c_{demanded}(s)$ ist. Jeder Knoten kann nun basierend auf dieser Information entscheiden, ob ein Service aktiviert oder deaktiviert werden muss.

6.4.2 Entscheidungsprozess

Um die Entscheidung über die Aktivierung bzw. Deaktivierung eines Dienstes zu treffen muss ein Knoten n zunächst die Abdeckung des Dienstes s in der direkten Nachbarschaft berechnen. Die aktuell erreichte lokale Abdeckung für einen Service s in der Nachbarschaft des Knotens n wird als $c_{local}(s, n)$ definiert. Jeder Knoten n berechnet $c_{local}(s, n)$ und vergleicht das Ergebnis mit der global geforderten Abdeckung $c_{demandedl}(s)$. Dabei sei $N(n)$ die Anzahl der direkten (1-hop) Nachbarn des Knotens n. Zu der Anzahl der Nachbarn wird der Knoten n als weiterer Knoten addiert. $S_s(n)$ sei die Anzahl der direkten Nachbarn des Knotens n, die den Service s aktiviert haben. $S_s(n)$ beinhaltet

dabei den Knoten n, wenn dieser den Service s ebenfalls aktiviert hat. Die aktuelle lokale Abdeckung für einen Service s und den Knoten n errechnet sich somit aus:

$$c_{local}(s,n) = \frac{S_s(n) * 100}{N(n) + 1} \tag{6.1}$$

In dieser Berechnung wird der Quotient mit 100 multipliziert, um eine Prozentangabe zu erhalten. Der Entscheidungsprozess, der auf jedem Knoten ausgeführt wird, basiert auf dieser lokalen Abdeckung $c_{local}(s,n)$ des jeweiligen Knotens n. Die Funktionsweise von *DySSCo* ist in Algorithmus 6.1 angegeben. Dabei reagiert der *DySSCo*-Algorithmus auf drei verschiedene Ereignisse: Ablauf des Intervalls t (Zeile 1-4), empfangene Nachrichen (Zeile 5-7) und auf abgelaufene Verzögerungszeiten (Zeile 8-25).

Der *DySSCo*-Algorithmus sendet in einem festgelegten Intervall t eine *DySSCo*-Nachricht und aktualisiert seine Nachbarschaftsliste (Zeile 1-4).

Wann immer eine *DySSCo*-Nachricht empfangen wird, wählt der empfangende Knoten eine zufällige Verzögerungszeit (engl.: *random backoff time*) $t_{backoff}$ (Zeile 5-7). Die zufällige Verzögerungszeit liegt zwischen 0 und der maximalen Verzögerungszeit $t_{max\_backoff}$. Nach Ablauf dieser Zeit wird eine Rückruffunktion aufgerufen. Diese Verzögerung verhindert, dass Geräte gleichzeitig reagieren, was zu ständigen simultanen Aktivierungen und Deaktivierungen führen würde. Wenn die Verzögerungszeit abläuft und die Rückruffunktion aufgerufen wird (Zeile 8), prüft der Knoten n die aktuelle lokale Abdeckung $c_{local}(s,n)$ für jeden Service s (Zeile 10-22).

Ist die aktuelle lokale Abdeckung $c_{local}(s,n)$ geringer als die global geforderte Abdeckung $c_{demanded}(s)$ für einen Dienst s und hat der Knoten n diesen Dienst nicht aktiviert, so aktiviert der Knoten n diesen Dienst s (Zeile 12).

Ist $c_{local}(s,n)$ größer als die geforderte Abdeckung $c_{demanded}(s)$ für den Dienst s und hat n diesen Dienst s bereits aktiviert, so berechnet n die erreichbare Abdeckung $c_{reachable}(s,n)$. Die erreichbare Abdeckung $c_{reachable}(s,n)$ berechnet sich dabei wie die aktuelle Abdeckung $c_{local}(s,n)$ unter der Annahme, dass der Dienst s auf dem Knoten n deaktiviert wurde. Nur wenn $c_{reachable}(s,n)$ noch immer größer gleich der global geforderten Abdeckung $c_{demanded}(s)$ für den Dienst s ist, deaktiviert n den Dienst s (Zeile 15-20).

Immer wenn sich der Status eines Dienstes auf einem Knoten ändert, wird sofort eine zusätzliche *DySSCo*-Nachricht gesendet, um die direkten Nachbarn über die aktuelle Abdeckung der Dienste zu benachrichtigen (Zeile 23-24).

Änderungen der aktuellen Abdeckung in einer Nachbarschaft eines Knotens beeinflussen die Abdeckungen in umliegenden Nachbarschaften. Bei Aktivierungen von Diensten wird die Information über die benötigte Abdeckung eines Dienstes durch die *DySSCo*-Nachrichten in neue Nachbarschaften weitergeleitet. Dadurch reagieren weitere Knoten mit der Aktivierung von Diensten und es wird eine globale Abdeckung für einen Dienst im gesamten Netz erreicht. Weiterhin kann so das Netz die Abdeckung auch bei Topologieänderungen aufrechterhalten.

Algorithmus 6.1 *DySSCo*-Algorithmus

```
1: on timeout callback                                   // Intervall t abgelaufen
2: sendBeacon()                                          // sende DySSCo-Nachricht
3: updateNeighborList()                                  // aktualisiere die Anzahl direkter Nachbarn
4: register timeout callback                             // registriere Rückruf in t für nächste DySSCo-Nachricht

5: on receive DySSCo beacon                              // Empfang einer DySSCo-Nachricht
6: updateServiceList()                                   // aktualisiere Liste known_services
7: register/reset backoff callback                       // registriere/ersetze Rückruf in zufälliger Zeit t_backoff

8: on backoff callback                                   // Backoff-Zeit t_backoff ist abgelaufen
9: changed = false
10: for each s in known_services do                      // iteriere über alle bekannten Dienste
11:     if c_local(s) < c_demanded(s) ∧ s inactive then
12:         activate(s)                                  // aktiviere Dienst s
13:         changed = true
14:     else
15:         if c_local(s) > c_demanded(s) ∧ s active then
16:             if c_reachable(s) ≥ c_demanded(s) then
17:                 deactivate(s)                        // deaktiviere Dienst s
18:                 changed = true
19:             end if
20:         end if
21:     end if
22: end for
23: if changed == true then
24:     sendBeacon()                                     // sende DySSCo-Nachricht
25: end if
```

6.4.3 Einstellungen und Parameter

Der *DySSCo*-Algorithmus hat drei verschiedene Parameter: das Intervall t der Nachrichten, die maximale Verzögerungszeit $t_{max\_backoff}$ und die geforderte globale Abdeckung $c_{demanded}(s)$ eines Dienstes. Die Wahl dieser Parameter wirkt sich auf das Verhalten des Algorithmus aus und wird im Folgenden näher betrachtet.

Anhand der *DySSCo*-Nachrichten, welche in einem Intervall von t gesandt werden, aktualisiert ein Knoten die Größe seiner Nachbarschaft und bestimmt die aktuelle lokale Abdeckung der Dienste. Die Wahl von t ist daher abhängig von Netzwerkeigenschaften wie Mobilität, Nachrichtenaufkommen sowie der Frage, ob Dienste lediglich aktiviert oder tatsächlich migriert werden. Es ist leicht ersichtlich, dass mit einem niedriger gewählten t die Knoten schneller auf Veränderungen in der Netztopologie reagieren. Der Nachteil dabei ist, dass sich dadurch das Nachrichtenaufkommen erhöht. Dies geschieht insbesondere dann, wenn ganze Dienste repliziert und von einem Knoten zum anderen migriert werden. Zusätzlich führen durch ein niedrig gewähltes t temporäre kurzzeitige Schwankungen in den Nachbarschaften zu Statusänderungen auf den Knoten. Es kann daher keine allgemeine Aussage getroffen werden, welche Größe von t sinnvoll ist, da dies direkt von dem Szenario sowie den Anforderungen seitens der Anwendung abhängt.

Die Verzögerungszeit $t_{backoff}$ verhindert, dass Knoten simultan auf eine Veränderung in der lokalen Abdeckung reagieren. Um das Problem zu verdeutlichen, betrachten wir drei Knoten, die einen vollständigen Graphen bilden, d.h. jeder kann mit jedem der 2 anderen Knoten direkt kommunizieren. Aktiviert ein Knoten einen Dienst s mit der geforderten globalen Abdeckung $c_{demanded}(s) = 50\,\%$ und sendet daraufhin eine *DySSCo*-Nachricht, so würden die beiden anderen Knoten ohne Verzögerungszeit gleichzeitig den Dienst s ebenfalls aktivieren. Diese Entscheidung treffen die Knoten, da die aktuelle von ihnen „gesehene“ lokale Abdeckung $c_{local}(s, n)$ bei $\approx 33\,\%$ liegt. Im nächsten Schritt würden jedoch alle Knoten erkennen, dass die lokale Abdeckung nun bei $100\,\%$ liegt. Als Folge würden ohne die Verzögerungszeit nun alle drei Knoten den Dienst s wieder deaktivieren. Dies kann zu einer unendlichen Folge von Aktivierungen und Deaktivierungen führen. Aus diesem Grund ist eine zufällige Verzögerungszeit auf den Knoten notwendig. Aus Skalierungsgründen ist es sinnvoll, die maximale Verzögerungszeit $t_{max\_backoff}$ in Abhängigkeit von der Nachbarschaftsgröße eines Knotens zu wählen. Je größer die Nachbarschaft eines Knotens, desto größer die maximale Verzögerungszeit $t_{max\_backoff}$, um die Wahrscheinlichkeit gleichzeitiger Reaktionen der Knoten zu verringern.

Die Wahl der geforderten globalen Abdeckung $c_{demanded}(s)$ für einen Dienst s ist abhängig von den Anforderungen des Applikationsszenarios an den Dienst. Da der *DySSCo*-Algorithmus versucht, die geforderte Abdeckung in jeder lokalen Nachbarschaft mindestens zu erreichen, ist die minimal zu erzielende Abdeckung begrenzt. In einer lokalen Nachbarschaft eines Knotens n der Größe $N(n)$ ist die minimal zu erreichende Abdeckung $1/(N(n)+1)$. Diese ist genau dann erreicht, wenn ein Knoten in der Nachbarschaft von n den Dienst anbietet. Dabei ist der Sonderfall, dass $c_{demanded}(s) = 0\,\%$ gilt, von der Betrachtung ausgenommen. Durch diese Begrenzung des lokalen Minimums ist auch die global erzielte Abdeckung in Abhängigkeit von der Netztopologie nach unten begrenzt.

In der aktuellen Version des *DySSCo*-Algorithmus werden nur die aktivierten Dienste in der Nachricht versandt. Bei einer gewählten globalen Abdeckung für einen Dienst s, für die gilt $c_{demanded}(s) \leq 1/(N(n)+1)$ würde kein weiterer Knoten in der Nachbarschaft von n diesen Dienst aktivieren. Somit sendet kein weiterer Knoten außer n die Information über den aktivierten Dienst. Die Information verbreitet sich nicht weiter. Diese Beschränkung kann jedoch durch das Versenden aller geforderten Abdeckungen für die verschiedenen Dienste in der *DySSCo*-Nachricht aufgehoben werden. Die Information, ob der Dienst auf dem sendenden Knoten aktiv ist, ist dann über eine zusätzliches Datum für jeden Service in der Nachricht enthalten.

6.5 Graphentheoretische Betrachtungen

Der *DySSCo*-Algorithmus basiert auf einfachen Entscheidungen, die jeder Knoten basierend auf lokalen Informationen trifft. Dabei ergeben sich zwei Fragestellungen: (1) Gibt es für jedes Netz mindestens eine stabile Verteilung von Diensten? (2) Erreicht der *DySSCo*-Algorithmus eine der stabilen Verteilungen?

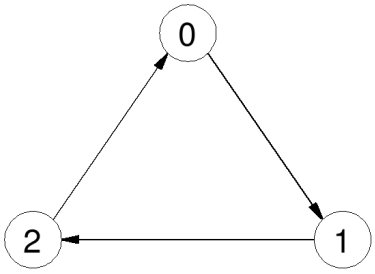


Abbildung 6.1: Graph, für den keine stabile Dienstverteilung mit *DySSCo* existiert

Knotenzahl	3	4	5
nicht-isomorphe Graphen	16	218	9608

Tabelle 6.2: Anzahl der nichtisomorphen Graphen für verschiedene Knotenzahlen

Die erste Fragestellung zielt darauf ab, ob es eine Verteilung eines Dienstes s auf den Knoten gibt, so dass kein Knoten unter der Annahme einer festen Topologie den Status des Dienstes s verändern möchte. Ist dies nicht erfüllt, so würde der *DySSCo*-Algorithmus ständig den Dienst s auf einzelnen Knoten aktivieren und deaktivieren. Es kann kein stabiler Zustand erreicht werden.

Die zweite Fragestellung setzt voraus, dass mindestens ein stabiler Zustand existiert. Es gilt zu beantworten, ob aus einer beliebigen Ausgangssituation heraus, aus welcher der Algorithmus initiiert wird, einer der stabilen Zustände erreicht wird. Wäre dies nicht der Fall, so würde der Algorithmus ebenfalls den Dienst auf verschiedenen Knoten aktivieren und deaktivieren.

6.5.1 Existenz stabiler Verteilungen

Es lässt sich zeigen, dass es Graphen gibt, für die keine stabile Verteilung der Dienste existiert. Ein einfaches Beispiel ist hierbei ein gerichteter zyklischer Graph, wie in Abbildung 6.1 dargestellt ist. Unter der Annahme, dass alle der drei Knoten über die Information verfügen, einen Dienst s mit einer geforderten Abdeckung $c_{demanded}(s) = 50\,\%$ zu verteilen, lässt sich keine stabile Verteilung finden. Aktiviert Knoten 0 den Service s, so deaktiviert Knoten 1 den Service, da die lokale Abdeckung von $c_{local}(s, 1)$ damit genau $50\,\%$ beträgt. Daraufhin aktiviert dann Knoten 2 den Service s, um die lokale Abdeckung zu erfüllen. In der Folge deaktiviert jedoch Knoten 0 den Service s, was wiederum eine Aktivierung auf Knoten 1 bedingt. Man erhält eine unendliche Folge von Aktivierungen und Deaktivierungen.

Um die praktische Relevanz der Graphen ohne stabile Verteilung einzuschätzen, sind alle nicht-isomorphen Graphen (Graphen, die auch bei einer beliebigen Vertauschung der Knotennummern in ihrer Struktur unterschiedlich sind) mit den Knotenzahlen $n = 3$,

$c_{demanded}(s)$ (in %)	Knotenzahl 3	4	5
25	1 (6 %)	12 (6 %)	522 (5 %)
33	1 (6 %)	11(5 %)	507 (5 %)
50	1 (6 %)	11 (5 %)	507 (5 %)
66	0 (0 %)	2 (<1 %)	357 (4 %)
75	0 (0 %)	0 (0 %)	23 (<1 %)

Tabelle 6.3: Prozentualer Anteil von Graphen, für die es keine stabile Dienstverteilung mit *DySSCo* gibt, für verschiedene Abdeckungen

$n = 4$ und $n = 5$ untersucht worden. Die Anzahl der nicht-isomorphen Graphen für die gegebenen Knotenzahlen sind in Tabelle 6.2 angegeben. Es wurde für alle möglichen Verteilungen der Dienste auf den Graphen untersucht, ob ein Knoten unter der gegebenen Verteilung den Status des Dienstes ändert. Die Auswertung aller Graphen in Tabelle 6.3 zeigt, dass es bei den untersuchten Graphen für bis zu 6 % der Graphen keine stabile Verteilung der Dienste gibt. Die betroffenen Graphen lassen sich nicht einer gemeinsamen Unterklasse der nicht-isomorphen Graphen zuordnen (beispielsweise nach [88]), enthalten jedoch meist gerichtete Zyklen.

Da diese Ergebnisse nur eine stichprobenartige Bedeutung haben, wird die Existenz sowie die Konvergenz im folgenden Abschnitt für die Teilmenge der zyklenfreien ungerichteten Graphen bewiesen. Die praktische Relevanz der Graphen ohne stabile Verteilung und die Tauglichkeit des *DySSCo*-Algorithmus für praktische Anwendungen wird in dem darauffolgenden Kapitel 6.6 mittels einer Evaluation mit 20 *Pacemate*-Sensorknoten untersucht.

6.5.2 Existenz und Konvergenz in ungerichteten zyklenfreien Graphen

Die alleinige Existenz einer stabilen Dienstverteilung genügt nicht, um eine Aussage über die Funktion des *DySSCo*-Algorithmus formulieren. Es muss außerdem bewiesen werden, dass der *DySSCo*-Algorithmus eine stabile Verteilung erreicht. Diese Eigenschaft des Erreichen eines stabilen globalen Endzustandes ohne zentrale Kontrolle wurde von Dijkstra bereits 1974 als *selbststabilisierend* (engl.: *self-stabilizing*) [14] definiert. Im Folgenden wird bewiesen, dass der *DySSCo*-Algorithmus für ungerichtete zyklenfreie Graphen ein selbststabilisierender Algorithmus ist.

Modellierung von Netzen

Wie bereits gezeigt kann der *DySSCo*-Algorithmus nicht für alle Netztopologien stabile Verteilungen erzeugen. Die Beweisführung beschränkt sich daher auf die ungerichteten zyklenfreien Graphen. Dabei wird sowohl die Existenz von stabilen Verteilungen sowie die Eigenschaft von *DySSCo*, einen stabilen Endzustand zu erreichen, bewiesen.

Algorithmus 6.2 Modellierung eines Algorithmus über Regeln

⟨Vorbedingung⟩ ⟶ ⟨Aktion⟩;
.
.
.
⟨Vorbedingung⟩ ⟶ ⟨Aktion⟩;

In einem verteilten System und insbesondere in einem Sensornetz können Aktionen gleichzeitig auf verschiedenen Komponenten (Knoten im Sensornetz) ausgeführt werden. Shukla et al. [96] führen dazu drei Modelle ein, die die Sequenz von Aktionen im Netz beschreiben: *central deamon*, *maximal parallelism* und *restricted parallelism*. Der *central deamon* beschreibt dabei das Modell einer zentralen Organisationseinheit, die immer nur eine Aktion eines Knotens im Netz zu einem Zeitpunkt erlaubt. Im Gegensatz dazu erlaubt das Modell des *maximal parallelism*, dass zu jedem Zeitpunkt jeder Knoten im Netz eine Aktion durchführen kann. Im Modell des *restricted parallelism* wird lediglich einer Teilmenge der Knoten im Netz ermöglicht, zur gleichen Zeit eine Aktion durchzuführen.

In Kapitel 6.4.3 wurde die Verzögerungszeit beschrieben, die die Wahrscheinlichkeit verringert, dass Knoten gleichzeitig auf eine *DySSCo*-Nachricht reagieren. Es wurde gezeigt, dass das simultane Reagieren zu einem endlosen Alternieren in den Zuständen führt. Durch dass Einführen der zufälligen Verzögerungszeit, die sich in Abhängigkeit von der Nachbarschaftsgröße ergibt, wird näherungsweise das Modell des *restricted parallelism* realisiert. Somit reagiert in einer Nachbarschaft mit großer Wahrscheinlichkeit immer nur ein Knoten. Das Modell des *restricted parallelism* wird daher der Beweisführung zu Grunde gelegt.

Modellierung von Algorithmen

Der *DySSCo*-Algorithmus ist ein sogenannter *uniformer* verteilter Algorithmus, da er auf allen Knoten gleichermaßen abläuft. Das Programm, das einen *uniformen* verteilten Algorithmus auf einem Knoten realisiert, kann dabei, wie in Algorithmus 6.2, als eine Menge von Regeln (engl.: *rules*) dargestellt werden. Die Regeln setzen sich dabei aus einer Vorbedingung (engl.: *precondition* oder auch *guard*) und einer Aktion (engl.: *action*) zusammen. Die Vorbedingung ist dabei eine boole'sche Funktion, die wahr oder falsch zurückliefert. Diese Funktion basiert dabei auf Informationen über den Status des Knotens und über den Status der Nachbarknoten. Können diese Vorbedingungen deterministisch berechnet werden, so gilt der gesamte verteilte Algorithmus als deterministisch. Wenn ein Knoten aktiv wird, prüft er, ob eine der Vorbedingungen erfüllt ist, und führt dann die entsprechende Aktion aus. Dabei kann der Knoten bei mehreren zutreffenden Bedingungen lediglich eine Aktion ausführen.

Algorithmus 6.3 Modellierung des *DySSCo*-Algorithmus über Regeln

1. $\langle a(s,n) = 0 \wedge c_{local}(s,n) < c_{demanded}(s) \rangle \longrightarrow \langle a(s,n) := 1 \rangle$;

2. $\langle a(s,n) = 1 \wedge c_{reachable}(s,n) \geq c_{demanded}(s) \rangle \longrightarrow \langle a(s,n) := 0 \rangle$;

Beweis von Existenz und Konvergenz

Um den Beweis für die Existenz einer stabilen Lösung und die Konvergenz des *DySSCo*-Algorithmus durchzuführen, modellieren wir den Algorithmus über seine Regeln. Dabei beschreibt $a(s,n)$ den Aktivierungszustand des Dienstes s auf dem Knoten n. Gilt $a(s,n) = 1$, ist der Dienst auf dem Knoten aktiviert. Gilt $a(s,n) = 0$, so ist der Dienst auf dem Knoten deaktiviert. Die Modellierung ist in Algorithmus 6.3 angegeben.

Satz 6.1. *In einem ungerichteten zyklenfreien Graphen findet der* DySSCo-*Algorithmus unter der Vorgabe des Modells des restricted parallelism einen stabilen Zustand.*

Beweis. Um die stabilen Zustände, in denen der *DySSCo*-Algorithmus keiner Veränderung der Zustände mehr bewirkt, identifizieren zu können, erweitern wir die Regeln so, dass alle möglichen Vorbedingungen dargestellt sind. Wie in Algorithmus 6.4 aufgeführt, erfolgen auf die erweiterten Regeln 3 und 4 keine Aktionen.

Algorithmus 6.4 Erweiterte Modellierung des *DySSCo*-Algorithmus über Regeln

1. $\langle a(s,n) = 0 \wedge c_{local}(s,n) < c_{demanded}(s) \rangle \longrightarrow \langle a(s,n) := 1 \rangle$;

2. $\langle a(s,n) = 1 \wedge c_{reachable}(s,n) \geq c_{demanded}(s) \rangle \longrightarrow \langle a(s,n) := 0 \rangle$;

3. $\langle a(s,n) = 0 \wedge c_{local}(s,n) \geq c_{demanded}(s) \rangle \longrightarrow \langle keineAktion \rangle$;

4. $\langle a(s,n) = 1 \wedge c_{reachable}(s,n) < c_{demanded}(s) \rangle \longrightarrow \langle keineAktion \rangle$;

Der Knoten ist unter den Vorbedingungen in Regel 3 und 4 stabil. Für einen beliebigen Knoten n betrachtet man zwei Fälle:

1. Fall: Der Knoten n reagiert auf eine *DySSCo*-Nachricht. Dabei ist auf Knoten n die Vorbedingung 1 erfüllt. Aufgrund des Modells des *restricted parallelism* reagiert der Knoten n als einziger in der Nachbarschaft von n. Knoten n aktiviert den Dienst s und sendet sofort eine *DySSCo*-Nachricht. Somit ist $a(s,n)$ auf 1 gesetzt.

Betrachtet man einen beliebigen Knoten m, der direkter Nachbar von n ist, so ist n im ungerichteten Fall auch ein Nachbar von m. Für m erhöht sich die Abdeckung in der Nachbarschaft. Dadurch, dass der Graph zyklenfrei ist, hat m nur durch die direkte Nachbarschaftsbeziehung zu n Auswirkungen auf die lokale Abdeckung für n. Für m betrachtet man ebenfalls die verschiedenen Fälle. Dabei sind die Fälle von Interesse,

wo Vorbedingungen der Regel 1 (1. Fall a)) oder Vorbedingung der Regel 2 (1. Fall b)) erfüllt sind, da diese wiederum eine Aktion bedingen, die die Abdeckung beeinflusst.

1. Fall a): Wenn für Knoten m trotz der Erhöhung der lokalen Abdeckung $c_{local}(s, m)$ durch n diese Abdeckung immer noch zu gering ist, so ist die Vorbedingung der Regeln 1 oder 4 erfüllt, je nach Status des Dienstes auf m. Ist der Dienst aktiviert (also $a(s, m) = 1$), so tritt Regel 4 in Kraft, die keine Aktion bedingt. Beide Knoten behalten den aktuellen Aktivierungsstatus. Ist der Dienst auf m deaktiviert, so gilt die Vorbedingung der Regel 1 für m und m aktiviert den Dienst und setzt $s(m)$ auf 1.

Durch den zu Grunde gelegten ungerichteten Graphen erhöht sich dadurch auch die lokale Abdeckung für den Knoten n. Für den Knoten n kann nun lediglich die Vorbedingung der Regeln 2 oder 4 gelten, wobei die Anwendung der Regel 4 keine Aktion und somit keine Änderung der Abdeckung nach sich führt. Ist die Vorbedingung der Regel 2 für n erfüllt, so deaktiviert der Knoten n den Dienst s und setzt $a(s, n)$ auf 0. Für Knoten m reduziert sich somit die lokale Abdeckung $c_{local}(s, m)$. Da $a(s, m)$ bereits auf 1 steht, wendet Knoten m lediglich Regel 4 an, was zu keiner Veränderung führt. Beide Knoten behalten den aktuellen Aktivierungsstatus für den Dienst s.

1. Fall b): Ist für den Knoten m nach der Erhöhung der lokalen Abdeckung durch den Knoten n die Abdeckung zu hoch, so treten je nach Aktivierungsstand des Dienstes auf m die Regeln 2 oder 3 in Kraft. Hierbei bedingt Regel 3 keine weitere Aktion und beide Knoten behalten ihren aktuellen Status. Ist die Vorbedingung der Regel 2 erfüllt, so deaktiviert m den Dienst s. Durch den ungerichteten Graphen verringert sich die lokale Abdeckung $c_{lokal}(s, n)$ wiederum. Durch die anfangs ausgeführte Regel 1 auf dem Knoten n gilt nun die Vorbedingung für Regel 4 für Knoten n, die keine Aktion bedingt. Beide Knoten behalten den aktuellen Aktivierungsstatus für den Dienst s.

2. Fall: Der Knoten n reagiert auf eine *DySSCo*-Nachricht. Dabei ist auf Knoten n Vorbedingung 2 erfüllt. Diese Betrachtung erfolgt analog zu Fall 1.

In beiden Fällen findet der *DySSCo*-Algorithmus auf beliebigen Knoten einen Status für einen Dienst s, so dass unter der Gegebenheit des *restricted parallelism* und eines ungerichteten zyklenfreien Graphens der Status des Dienstes auch bei Veränderungen in der Nachbarschaft nicht mehr durch *DySSCo* verändert wird. Der *DySSCo*-Algorithmus findet somit einen stabilen Zustand und ist daher selbststabilisierend. □

6.6 Evaluation

In der graphentheoretischen Betrachtung konnte gezeigt werden, dass *DySSCo* zum einen nicht in allen möglichen Graphen eine Lösung finden kann. Zum anderen wurde bewiesen, dass *DySSCo* in zyklenfreien ungerichteten Graphen, zu denen beispielsweise Ketten und Bäume zählen, gegen einen stabilen Zustand konvergiert. Um nun die Praxistauglichkeit des *DySSCo*-Algorithmus zu demonstrieren, wurde keine Simulation, sondern ein Experiment mit den *Pacemate*-Sensorknoten durchgeführt. Jeder Knoten zeichnete dabei die Anzahl seiner Nachbarn, die lokal gemessene Abdeckung sowie den Status des

Services auf dem Knoten und die versendeten *DySSCo*-Nachrichten auf. Hierdurch lassen sich lokale Abdeckungen mit der global erzielten Abdeckung vergleichen. Zudem kann das vom *DySSCo*-Algorithmus erzeugte Nachrichtenaufkommen analysiert werden.

6.6.1 Aufbau und Parameter

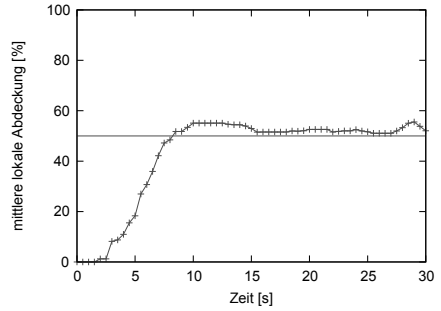
Die Messung wurde mit 20 *Pacemate*-Sensorknoten durchgeführt. Die *Pacemates* wurden dabei entlang eines Korridors platziert. Die durchschnittliche Nachbarschaftsgröße betrug sieben Knoten. Das Intervall t für die *DySSCo*-Nachrichten war auf $t = 3\,\mathrm{s}$ gewählt. Erhielt ein Knoten n über einen Zeitraum von 12 s keine Nachricht des Knoten m, so wurde der Knoten m aus der Nachbarschaftsliste des Knotens n entfernt. Die maximale Verzögerungszeit wurde auf $t_{max\_backoff} = 200\ \mathrm{ms} * (N(n) + 1)$ gesetzt, wobei $N(n)$ die Nachbarschaftsgröße von n bezeichnet. Die geforderte Abdeckung $c_{demanded}(s)$ für den Service s wurde in der Messung auf 50 % festgelegt. Auf Tastendruck wurde ein Service auf dem äußersten Knoten mit der geforderten Abdeckung $c_{demanded}$ aktiviert. Dadurch wurde die sukzessive Verteilung des Dienstes im Netz angestoßen.

6.6.2 Messergebnisse

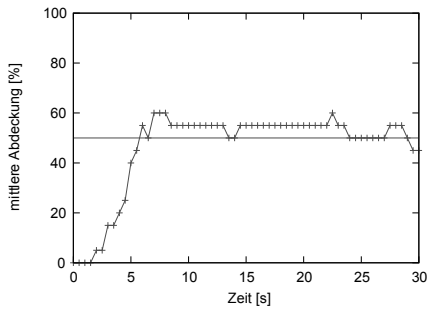
Basierend auf den aufgezeichneten Daten wurde die Funktionalität des *DySSCo*-Algorithmus evaluiert. Dabei wurden die ersten 30 s nach der Aktivierung des Dienstes s auf dem ersten Knoten ausgewertet. Neben der lokalen Abdeckung $c_{local}(s, n)$ für jeden Knoten wurde auch die globale Abdeckung c_{global} ausgewertet, die sich aus dem Status des Dienstes s auf jedem Knoten berechnen ließ. Zusätzlich wurden die Nachbarschaftsgrößen über die Zeit und somit die Veränderungen der Nachbarschaften ausgewertet sowie die Anzahl der *DySSCo*-Nachrichten.

Abbildung 6.2(a) zeigt den Durchschnitt der lokalen von den Knoten berechneten Abdeckung $c_{local}(s, n)$. Es ist zu erkennen, wie die lokale gemessene Abdeckung am Anfang nach der Aktivierung des Dienstes am ersten Knoten schnell bis zu 55 % ansteigt. Der Graph zeigt, wie das Netz nach dem Überschwingen einen stabilen Zustand für die lokale Abdeckung knapp über der vorgegebenen Abdeckung von 50 % erreicht.

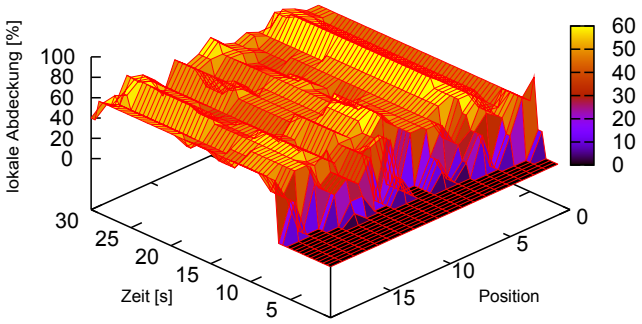
Die globale Abdeckung $c_{global}(s)$ bezeichnet den tatsächlich im Netzwerk vorhandenen Prozentsatz von Knoten, die den Dienst s anbieten. Abbildung 6.2(b) zeigt die von *DySSCo* erzeugt globale Abdeckung. Die globale Abdeckung steigt ebenfalls zu Beginn der Messung schnell an und stabilisiert sich bei 55 %. Aus dem Vergleich des Verlaufs der globalen Abdeckung mit dem der lokalen Abdeckung ist zu erkennen, wie das Bestreben des *DySSCo*-Algorithmus, die lokale Abdeckung an die geforderte Abdeckung anzupassen, die globale Abdeckung beeinflusst. Die Schwankungen der globalen Abdeckung um 5 % Schritte ergeben sich aus der Tastache, dass ein Knoten aus einer Gesamtknotenzahl von 20 bereits zu 5 % der Abdeckung beiträgt. So führen zwölf Knoten, die einen Dienst s aktiviert haben, bereits zu einer globalen Abdeckung von 60 %.



(a) Lokale Abdeckung $c_{local}(s, n)$ gemittelt über die Knoten



(b) Globale Abdeckung des Dienstes

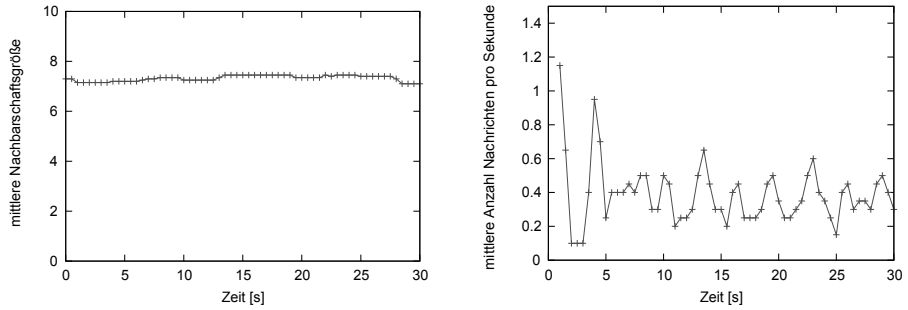


(c) Entwicklung der Abdeckung über die Zeit und Position der Knoten

Abbildung 6.2: Ergebnisse für das Experiment mit 20 *Pacemates*

Abbildung 6.2(c) veranschaulicht, wie sich der Dienst vom Knoten an der Position 0 über die Zeit in dem Netz verbreitet. Auf der Applikate (z-Achse) ist dabei die von den einzelnen Knoten gesehene lokale Abdeckung $c_{local}(s, n)$ dargestellt. Der dargestellte Verlauf beginnend bei der Position 0 verdeutlicht nochmals den globalen Prozess der lokalen Anpassungen.

Die Variationen in der Abdeckung über die Zeit lassen sich durch Veränderungen in den Nachbarschaften erklären. Aus Abbildung 6.3(a) lässt sich erkennen, dass die durchschnittliche Nachbarschaftsgröße über der Zeit Schwankungen unterliegt. Sofern die Veränderung der Nachbarschaft gemäß des *DySSCo*-Algorithmus keine Veränderung des Status auf den einzelnen Knoten auslöst, so tragen die einzelnen Knoten in einer kleineren Nachbarschaft dennoch stärker zur Abdeckung bei. Inbesondere bei kleinen Nachbarschaftsgrößen steigt dabei die lokale Abdeckung stark an. Betrachtet man beispielsweise ein Szenario mit einem Knoten und drei Nachbarknoten, so haben dort bei einer geforderten Abdeckung von 50 % zwei Knoten den Dienst aktiviert. Die Abdeckung von 50 % wird somit genau erfüllt. Wird nun ein Knoten, der den Dienst deaktiviert hat, aus der Nachbaschaft entfernt, so ergibt sich eine lokale Abdeckung von $\approx 66\,\%$. Der *DySSCo*-Algorithmus verändert den Status des Dienstes auf den Knoten nicht, da keine bessere



(a) Durchschnittliche Nachbarschaftsgröße (b) Mittleres Nachrichtenaufkommen pro Knoten

Abbildung 6.3: Nachbarschaftsgrößen und Nachrichtenaufkommen für das Experiment mit 20 *Pacemates*

Annäherung an die Erfüllung der geforderten 50 % erreicht werden kann. Die Veränderungen der Nachbarschaftsverhältnisse ist auf Störungen des Funkmediums zurückzuführen, die unter anderem von Personen hervorgerufen wurden, die sich während der Messung zwischen den Knoten bewegten.

Um die Kosten des *DySSCo*-Algorithmus im Bezug auf das Nachrichtenaufkommen einschätzen zu können, wurde während des Experimentes die Anzahl der *DySSCo*-Nachrichten aufgezeichnet. Abbildung 6.3(b) zeigt die durchschnittliche Anzahl der pro Knoten und pro Sekunde versandten *DySSCo*-Nachrichten über den Verlauf des Experimentes. Es ist zu erkennen, dass zu Beginn der Messung ein sehr hohes Nachrichtenaufkommen entsteht. Dies ist darauf zurückzuführen, dass nach der Aktivierung des durch *DySSCo* zu verteilenden Dienstes viele der Knoten den Status des Dienstes ändern und somit zusätzliche *DySSCo*-Nachrichten versenden, um die Nachbarschaft sofort über die Statusänderung zu informieren. In der Folge pendelt das Nachrichtenaufkommen konstant um 0,4 Nachrichten pro Sekunde pro Knoten. Das bedeutet, dass in drei Sekunden 1,2 Nachrichten versandt werden. Dies entspricht näherungsweise dem gewählten Intervall von $t = 3$ s, in dem der *DySSCo*-Algorithmus Nachrichten versendet.

Das Experiment zeigt, dass *DySSCo* die geforderte Abdeckung für einen Dienst in einem realen Netz erreicht und auch bei Schwankungen in den Nachbarschaften aufrechterhält. Gleichzeitig bleibt das durch *DySSCo* erzeugte Nachrichtenaufkommen im gemessenen Szenario näherungsweise konstant. Auch in weiteren Demonstrationen z.B. in [66] konnte die Funktion und Praxistauglichkeit von *DySSCo* für verschiedene Topologien, Abdeckungen und unter Mobilität gezeigt werden.

6.7 Ergebnis

In diesem Kapitel wurde auf die Notwendigkeit der automatischen und selbstorganisierenden Dienstverteilung in Sensornetzen eingegangen. Die automatische Dienstverteilung

ist dabei eine Methode, um die Dynamik des Netzes vor dem Anwendungsersteller zu verbergen und ihm eine Abstraktion zur Verfügung zu stellen, mittels derer die Anforderungen an eine Anwendung formuliert werden können. Es wurde zunächst auf existierende Ansätze eingegangen, die mittels einer Lösung des sogenannenten *Facility Location Problem* die optimale Anzahl von Replikaten eines Diensten sowie deren optimale Platzierung berechnen. Dabei sind die Lösungen abhängig von globalem Wissen, Wissen über die eigene Position oder zu Grunde liegenden Wegewahlverfahren.

Aus der Beobachtung heraus, dass nur eine der verwandten Arbeiten tatsächlich realisiert worden ist, und unter der Annahme, dass eine optimale Platzierung in einem inhärent dynamischen Umfeld wie einem Sensornetz nicht zielführend ist, hat der Autor den *DySSCo*-Algorithmus entwickelt. Der *DySSCo*-Algorithmus arbeitet mit vorgegebenen Abdeckungen für einzelne Dienste. Durch das Bestreben, diese Abdeckungen lokal immer aufrecht zu erhalten, erreicht *DySSCo* basierend auf lediglich lokalen Informationen und zwei einfach zu realisierenden Regeln, dass die Abdeckung auch global erreicht wird. *DySSCo* selbst ist dabei nicht auf andere Protokolle angewiesen und zudem auch in mobilen Szenarien einsetzbar.

Die graphentheoretische Betrachtung ergab, dass *DySSCo* nicht für alle Graphen einen stabilen Endzustand erreichen kann. Die betroffenen Graphen enthielten dabei meist gerichtete Zyklen und sind vermutlich die Ausnahme in Sensornetztopologien. Sie erscheinen daher für reale Einsätze nicht relevant. Aufgrund der stichprobenartigen Untersuchung kann jedoch keine allgemeingültige Aussage aus diesen Ergebnissen abgeleitet werden. Es ist jedoch zu bemerken, dass bei vielen Algorithmen zur Dienstverteilung sogar ungerichtete Netztopologien vorausgesetzt werden. Für *DySSCo* wurden hingegen auch gerichtete Topologien untersucht, bei denen lediglich 5 % der Graphen zu keiner stabilen Lösung führen. Es konnte zusätzlich bewiesen werden, dass der *DySSCo*-Algorithmus auf azyklischen ungerichteten Netztopologien stabile Endzustände besitzt und gegen diese konvergiert.

In Experimenten mit den *Pacemate*-Sensorknoten konnte zudem gezeigt werden, dass der *DySSCo*-Algorithmus auch in der Praxis einsetzbar ist.

In weiteren Arbeiten ist geplant, die *DySSCo*-Nachricht mit Zeitstempeln für die angegebene Abdeckung zu versehen, um die Abdeckung im Laufe der Anwendung neu setzen zu können. Eine solche Erweiterung bedarf jedoch eines Mechanismus, der die Zeitstempel eindeutig festlegt. Wird die Abdeckung für die Dienste an zentraler Stelle festgelegt, so kann dies mittels einfacher Sequenznummern realisiert werden. Erfolgt die Festlegung der Abdeckung dezentral, so wird eine globale Zeit benötigt, um global eindeutige Zeitstempel zu generieren. Hier könnte beispielsweise die logische Zeit von Lamport [51] verwendet werden.

Eine weitere Weiterentwicklung des *DySSCo*-Algorithmus wäre, dass die *DySSCo*-Nachrichten alle Dienste mit ihrem Status versenden. Konnten bisher Szenarien konstruiert werden, wo sich ein Dienst nicht global ausbreitet, da lediglich aktivierte Dienste in die *DySSCo*-Nachricht aufgenommen wurden, so kann dadurch in jedem Fall zugesichert werden, dass der Dienst sich global verteilt. Die Einschränkung für die minimale Abde-

ckung, dass diese größer als $1/(N(n)+1)$ für die globale Ausbreitung gewählt sein muss, würde dadurch aufgehoben.

Bei allen Erweiterungen muss jedoch berücksichtigt werden, dass die Grundeigenschaften des *DySSCo*-Algorithmus nämlich Einfachheit, Robustheit und Unabhängigkeit von anderen Protokollen bestehen bleiben.

7 GRAPE – Pfadunabhängiges Multi-hop-Routing

Das Problem der Wegewahl (engl.: *routing*) in sich spontan verbindenden Funknetzen wurde in dieser Arbeit bereits mehrfach angesprochen. Die Wegewahl löst das Problem, über welche Knoten im Netz Daten von einer Quelle (engl.: *source*) zu einer Senke (engl.: *destination*) gelangen. Abstrahiert man wie in Kapitel 2.1 dargestellt das Netz als Graph und legt diesem Modell zu Grunde, dass die Kommunikationsverbindungen bidirektional sind und keinerlei Schwankungen unterliegen, so kann das Problem der Wegewahl leicht graphentheoretisch gelöst werden. Eine solche Abstraktion ist für Funknetze und insbesondere für die Entwicklung von Wegewahlverfahren zu stark vereinfacht. Die Annahmen des einheitlichen kreisrunden Funkradius, des sogenannten *unit disk graph* Modells, sowie die zeitliche Konstanz sind nicht haltbar. Wie die Erfahrungen und dabei insbesondere die Verlustraten der Nachrichten aus den in Kapitel 2.1.3 exemplarisch aufgeführten technischen Umsetzungen von Sensornetzen zeigen, stellt die Wegewahl trotz vieler vorhandener Algorithmen noch immer ein großes Problem dar. Die Qualitäten der Links unterliegen Schwankungen. Zudem sind Links nicht immer bidirektional. In der Arbeit von Zhou et al. [130] wird anhand von Messungen sogar gezeigt, dass unidirektionale Links in Abhängigkeit von der Knotendistanz eher zur Regel gehören. Vielen Arbeiten zu Wegewahlalgorithmen liegen jedoch genau diese Annahmen zu Grunde. Erst komplexe Erweiterungen ermöglichen, dass Wegewahlalgorithmen unter realen Bedingungen funktionieren und unidirektionale und verschwindende Links oder auch Mobilität unterstützen.

Die grundlegende Problematik besteht darin, dass viele Algorithmen auf der statischen Graphenabstraktion entwickelt werden und Pfade über die Links zwischen Quelle und Senke pflegen. Dies bedeutet, dass man die statischen Konstrukte des Pfades und des Links auf eine in der Praxis dynamische Infrastruktur anwendet. Dies ist darin begründet, dass zunächst versucht wurde, existierende Lösungen aus kabelgebundenen Netzen in funkbasierte Netze zu übertragen.

In diesem Kapitel wird das im Rahmen der Arbeit entwickelte pfadunabhängige Routingprotokoll *GRAPE* (*Gradient based Routing for all PurposEs*) [62] vorgestellt. *GRAPE* wurde speziell für Funknetze entwickelt und nutzt die Eigenschaft des Funkmediums aus, dass alle direkten Nachbarn eine gesendete Nachricht hören. *GRAPE* arbeitet dabei nicht mit einzelnen Links und legt somit nicht die weiterleitenden Knoten fest, die ggf. nicht mehr erreichbar sind. Der Algorithmus ist dabei nicht abhängig von exakten Informationen über die Netztopologie, da diese nach ihrer Erhebung durch die physikalischen Eigenschaften des Funks bereits nicht mehr gültig sein können.

Im Folgenden wird im Rahmen einer Klassifikation ein Überblick über verschiedene Wegewahlverfahren gegeben. Dabei werden in dieser Arbeit ausschließlich sogenannte *Unicast*-Verfahren betrachtet, die als Datensenke genau einen Knoten adressieren und im Gegensatz zu *Multicast*-Verfahren nicht eine ganze Gruppe von Knoten adressieren. Basierend auf einer Bewertung der verwandten Arbeiten werden anschließend die Anforderungen an ein Wegewahlverfahren formuliert. Aus diesen Anforderungen wird der algorithmische Ansatz des *GRAPE*-Wegewahlverfahrens abgeleitet und im Detail vorgestellt. Im Anschluss wird die Implementation des *GRAPE*-Algorithmus skizziert. Abschließend wird der *GRAPE*-Algorithmus evaluiert. Die Ergebnisse werden zusammengefasst.

7.1 Verwandte Arbeiten

Im Bereich der Wegewahl in drahtlosen Netzen gibt es bereits viele Forschungsarbeiten. Dennoch ist das Problem der Wegewahl praktisch noch nicht gelöst. Im Folgenden werden Wegewahlverfahren klassifiziert und einzelne Vertreter jeder Klasse vorgestellt. Dabei wird außerdem auf die Probleme der einzelnen Verfahren hinsichtlich der Praxistauglichkeit eingegangen.

7.1.1 Klassifikationen von Wegewahlverfahren

Wegewahlverfahren für drahtlose Netze lassen sich auf verschiedene Arten klassifizieren. Eine weit verbreitete Klassifikation, über die Royer und Toh [90] in ihrer Arbeit einen Überblick geben, ist die Unterteilung in *proaktive*, *reaktive* und *hybride* Verfahren. Diese Unterteilung bezieht sich darauf, dass sich die Verfahren darin unterscheiden, wann die einzelnen Knoten die Informationen akquirieren, auf deren Basis die Wegewahl getroffen wird.

Proaktive Verfahren (engl.: *proactive* oder *table driven*) versuchen, aktuelle und konsistente Informationen auf jedem Knoten für jeden Zielknoten im Netz vorzuhalten. Bei diesen Verfahren werden diese Routinginformationen in sogenannten Routingtabellen (engl.: *routing tables*) gehalten. Bei Änderungen in der Topologie des Netzes werden entsprechende Informationen sofort durch das Netz propagiert, um die Einträge in Tabellen aktuell und konsistent zu halten. Diese Nachrichten, die von den Verfahren zu diesem Zweck versendet werden, werden als *Kontrollnachrichten* bezeichnet.

Reaktive Verfahren (engl.: *reactive*, *on demand* oder *source initiated*) akquirieren die Informationen erst dann, wenn eine Quelle einen Pfad zu einer Senke benötigt. Ist dies der Fall, so wird ein sogenannter *Route-Discovery-Prozess* gestartet, anhand dessen die aktuelle Route von der Quelle zur Senke im Netz gefunden wird. Ist eine Route gefunden, so wird versucht, diese aufrecht zu erhalten, bis die Senke nicht mehr erreichbar ist oder die Route nicht mehr benötigt wird.

Der Vergleich der beiden Verfahrensweisen macht deutlich, dass sich beide für verschiedene Szenarien besser bzw. schlechter eignen. Dabei spielen Netzparameter wie Knotenzahl

und Mobilität (Anzahl der Topologieänderungen), aber auch Applikationsparameter wie Verzögerungstoleranz bei der Auslieferung und Funkverkehrsmuster (beispielsweise: Datenaufkommen und Sendeintervalle) eine Rolle. Betrachtet man diese Parameter, wird schnell deutlich, dass proaktive Verfahren in großen Netzen große Routingtabellen im Speicher halten müssen. Zudem werden ständig Nachrichten mit Topologieinformationen versandt, auch wenn keine Daten seitens der Applikation übertragen werden. Reaktive Verfahren hingegen müssen, wenn zu versendende Daten vorliegen, zunächst den Route Discovery Prozess starten. Dies verzögert zum einen die Auslieferung der Daten. Zum anderen führt dies aber auch zu einer kurzzeitigen extremen Auslastung des Netzes, da während dieses Prozesses Anfragen zur Bestimmung des Pfades zur Senke durch das Netz gesandt werden. Reaktive Verfahren erzeugen jedoch keine Nachrichten, wenn keine Daten zu senden sind. Im Hinblick auf die Rahmenbedingungen in Sensornetzen, wo sowohl Speicher als auch Energie, die zum Senden benötigt wird, stark begrenzt sind, müssen bei der Auswahl eines reaktiven oder proaktiven Routingprotokolls die vom Netz und von der Applikation gegebenen Parameter genauestens berücksichtigt werden.

Hybride Verfahren versuchen, die Vorteile beider Verfahren zu kombinieren. So kann ein hybrider Algorithmus beispielsweise im lokalen Bereich eines Knoten proaktiv agieren, wobei für entfernte Knoten die Route reaktiv bestimmt wird.

Es lassen sich jedoch nicht alle Verfahren in diese genannten Klassen einordnen. Das Weiterleiten von Nachrichten von Quelle zur Senke mittels des sogenannten *Flutens* (engl.: *flooding*) durch das Netzwerk gehört zu keiner der obigen Klassen. Beim Fluten leitet jeder Knoten eine empfangene Nachricht weiter. Die Senke empfängt somit ebenfalls diese Nachricht und kann diese verarbeiten. Hier werden keine Routinginformationen erhoben, weder reaktiv noch proaktiv. Auch das im Folgenden vorgestellte *GRAPE* lässt sich nicht in eine der genannten Klassen einordnen. Daher stellt sich die Frage nach einer weiteren Klassifizierung. Dabei wird nicht darauf Bezug genommen, wann Routinginformationen akquiriert werden, sondern auf welchen Informationen die Wegewahlentscheidung getroffen wird. Im Folgenden wird auf *Distanzvektorverfahren*, *Link-State-Verfahren*, *Geographische Verfahren* und *gradientenbasierte Verfahren* eingegangen. Dennoch ist die Fragestellung, wann Routinginformationen gesammelt werden, insbesondere in dem *GRAPE*-Algorithmus zu einem späteren Zeitpunkt von Interesse.

7.1.2 Distanzvektorverfahren

Die sogenannten *Distanzvektorverfahren* (engl.: *distance-vector*) arbeiten auf Distanzen zu den möglichen Zielknoten. Jeder Knoten hält für jede Senke Distanzen mit den zugehörigen Nachbarknoten, sozusagen den Richtungsvektor, über den die Senke mit der angegebenen Distanz erreicht wird. Ein zu versendendes Datenpaket wird an den Nachbarn weitergeleitet, der die geringste Distanz zur Senke hat. In dieser Weise wird auf jedem Knoten die Entscheidung über den nächsten weiterleitenden Knoten neu getroffen, bis das Paket die Senke erreicht hat. Distanzvektorverfahren realisieren dabei einen verteilten *Bellman-Ford-Algorithmus* zur Berechnung der kürzesten Pfade auf einem gewichteten Graphen. Zu Beginn ist die Distanztabelle lediglich mit den bekannten

Zielknoten	Nächster Hop	Metrik (Distanz)	Sequenznummer
n_1	n_3	3	$SeqNr_1$
n_2	n_2	0	$SeqNr_2$
n_3	n_3	1	$SeqNr_3$
n_5	n_5	1	$SeqNr_5$
n_8	n_5	4	$SeqNr_8$
...	...	...	...

Tabelle 7.1: Beispielhafte vereinfachte DSDV Routingtabelle des Knotens n_2

Zielknoten	Metrik (Distanz)	Sequenznummer
n_1	3	$SeqNr_1$
n_2	0	$SeqNr_2$
n_3	1	$SeqNr_3$
n_5	1	$SeqNr_5$
n_8	4	$SeqNr_8$
...	...	...

Tabelle 7.2: Beispiel einer DSDV Nachricht des Knotens n_2 mit Zielknoten, Distanzen und Sequenznummern

Distanzen zu den direkten Nachbarn gefüllt. Es existieren verschiedene Ansätze, wie die Distanzinformationen verteilt und auf den Knoten aktuell und konsistent gehalten werden können. Dies kann sowohl in proaktiver als auch reaktiver Weise erfolgen. Es wird im Folgenden exemplarisch auf die Wegewahlverfahren DSDV und AODV als Beispiele für Distanzvektorverfahren eingegangen.

Destination-Sequenced Distance-Vector Routing (DSDV)

Das *Destination-Sequenced Distance-Vector Routing* (DSDV) [83] von Perkins und Bhagwat aus dem Jahr 1994 ist ein proaktives Distanzvektorverfahren. Jeder Knoten n_i hält eine Routingtabelle mit Distanzen zu Zielknoten sowie den zugehörigen Nachbarn, über den der jeweilige Zielknoten mit der gegebenen Distanz erreicht wird. Zusätzlich werden Sequenznummern gespeichert. Tabelle 7.1 zeigt eine beispielhafte Routingtabelle für einen Knoten n_2. In der Spezifikation von Perkins und Bhagwat werden zusätzlich noch Zeitstempel und sogenannte *flags* zu jedem Eintrag gehalten, um die Robustheit und Effizienz des Verfahrens zu steigern, in dem sich beispielsweise ein optimales Sendeintervall für Routinginformationen aus den Informationen ableiten lässt. Jeder Knoten versendet in diesen gegebenen Zeitintervallen proaktiv in seiner direkten Nachbarschaft die Informationen über die ihm bekannten Knoten, die ihm bekannte Distanz zu diesen Knoten sowie die zugehörigen Sequenznummern. Tabelle 7.2 zeigt eine solche beispielhafte Nachricht für einen Knoten n_2. Der Knoten n_2 ist dabei ebenfalls in der Nachricht

mit der Distanz 0 enthalten. Die Sequenznummern für einen Knoten n_i werden von dem Knoten n_i selber festgelegt. Die Nummern sind immer gerade und steigen mit jeder neu ausgesendeten Nachricht streng monoton an. Der Empfänger einer solchen Nachricht pflegt die empfangene Informationen in seine Routingtabelle ein. Dabei werden immer Einträge für den Knoten n_i mit höherer Sequenznummer $SeqNr_i$ in die Tabelle übernommen, da diese aktuelleren Informationen entsprechen. Die Metrik wird ebenfalls erhöht. Entspricht die Metrik der Anzahl der zurückgelegten Hops, so wird diese um 1 erhöht und für den Zielknoten in die Tabelle eingetragen. Bei Informationen für einen Knoten mit identischer Sequenznummer wird der Eintrag mit niedrigerer Metrik übernommen, um den günstigsten (kürzesten) Pfad zu erhalten. Erkennt ein Knoten n_i, dass ein Zielknoten n_j nicht mehr erreichbar ist, so versieht n_i dessen Distanzeintrag mit dem Wert ∞. Zudem erhöht n_i die Sequenznummer $SeqNr_j$ für den Knoten n_j um 1 und erzeugt somit eine ungerade höhere Sequenznummer für n_j.

Durch den Mechanismus des lokalen Verteilens aller bekannten Informationen werden die Informationen im gesamten Netz verteilt. DSDV erreicht somit, dass jeder Knoten den nächsten Nachbarn für den günstigsten Weg zu einer Senke kennt.

Ad-hoc On-Demand Distance-Vector Routing (AODV)

Das *Ad-hoc On-demand Distance-Vector Routing* (AODV) [84][82] von Perkins und Royer aus dem Jahr 1997 ist ein reaktives Distanzvektorverfahren. Dabei werden Routinginformationen nur von solchen Knoten gehalten und gepflegt, die auf gerade aktiv genutzten Pfaden liegen. Knoten pflegen somit keine Pfade zu anderen Knoten, wenn diese nicht miteinander kommunizieren. Die Ziele des Algorithmus sind, erstens Routingpakete nur dann zu versenden, wenn die Routinginformationen benötigt werden, zweitens zwischen lokalen direkten Nachbarschaftsinformationen und globalen Topologieinformationen zu unterscheiden und drittens Informationen über Topologieveränderungen nur an solche Knoten weiterzugeben, die diese Informationen benötigen.

Liegen dem Knoten n_i Daten vor, die an einen anderen Knoten n_j versandt werden sollen, so wird zunächst der Pfad bzw. die Route von der Quelle n_i zu der Senke n_j akquiriert. Der Knoten n_j sendet ein sogenanntes *Route-Request-Paket* (RREQ). Dieses Paket enthält die Quelladresse, eine Quellsequenznummer, eine Broadcast ID, die Adresse der Senke, eine Sequenznummer der Senke und einen Hopcount. Besitzt ein empfangender Knoten aktuelle Informationen über den Pfad zur Senke, so antwortet dieser mit einem sogenannten *Route-Reply-Paket* (RREP). Ist dies nicht der Fall, wird das RREQ-Paket weitergeleitet. Mittels einer sogenannten *packet history* wird ein RREQ-Paket, das anhand der Quelladresse sowie der Broadcast ID eindeutig identifiziert werden kann, nur einmal bearbeitet bzw. weitergeleitet.

Um ein RREP-Paket zurück zur Quelle senden zu können, merkt sich jeder Knoten, von welchem Knoten er das RREQ-Paket erhalten hat. Diese Information wird nach einem gewissen Zeitintervall wieder verworfen. Erreicht ein RREQ-Paket einen Knoten, der einen Pfad zur Senke besitzt, so vergleicht dieser zunächst die im Paket enthaltene

Sequenznummer der Senke mit der eigenen zur Senke abgespeicherten Sequenznummer. Ist die über das RREQ-Paket erhaltene Sequenznummer größer, so wird das RREQ weitergeleitet, da davon ausgegangen wird, dass die lokale Pfadinformation auf dem Knoten veraltet ist. Ist jedoch die Sequenznummer zur Quelle auf dem Knoten größer oder gleich, so antwortet der Knoten mit einem RREP-Paket, welches entlang des bekannten Pfads zur Quelle gesandt wird. Dabei merkt sich nun jeder Knoten, von welchem Knoten das RREP-Paket erhalten wurde zusammen mit der aktuellen Sequenznummer zu Senke. Somit ist den Knoten, die am Pfad teilnehmen, der aktuelle nächste Knoten auf dem Pfad zur Senke bekannt.

Die Knoten, die aktiv an dem Pfad beteiligt sind, senden regelmäßig sogenannte *Hello*-Nachrichten in ihre direkte Nachbarschaft. Dadurch kontrollieren die empfangenden Knoten die Existenz der verwendeten Links. Wird erkannt, dass ein Link im Pfad nicht mehr existiert, wird ein neues RREP von dem Knoten an alle Knoten des Pfades in Richtung Quelle gesandt mit höherer Sequenznummer für die Senke und einem unendlichen Hopcount. Somit kann die Quelle, sofern der Pfad weiter benötigt wird, einen neuen Route-Request starten.

Durch diesen Mechanismus realisiert AODV ein Verhalten, bei dem nur bei Bedarf Routinginformationen gesammelt werden und lediglich Knoten entlang eines genutzen Pfades Informationen über Topologieveränderungen austauschen.

7.1.3 Link-State-Verfahren

Bei den *Link-State-Verfahren* hält jeder Knoten ein Abbild der kompletten Netztopologie. Basierend auf diesen Informationen kann jeder Knoten den kürzesten Pfad zu einer Senke berechnen. Dies passiert über die lokale Berechnung auf einem Knoten mittels des *Dijkstra-Algorithmus* oder *Bellman-Ford-Algorithmus* zur Bestimmung der kürzesten Pfade. Der nächste Knoten für ein zu versendendes Datenpaket wird anhand der Lösung des kürzesten Weges zur Senke bestimmt. Diese Berechnung wird auf jedem weiterleitenden Knoten durchgeführt, bis das Paket die Senke erreicht hat.

Ein Beispiel für ein Link-State-Verfahren ist das *Optimized Link State Routing* (OLSR) von Jacquet et al. [40][8]. OLSR versendet proaktiv die Informationen über den Status der Links der einzelnen Knoten durch das gesamte Netz. Dabei wird bei OLSR jedoch nur eine Teilmenge der Links eines Knotens versandt. Zudem werden nur von ausgewählten Knoten Link-State-Informationen durch das gesamte Netzwerk geflutet. Dadurch wird die Anzahl der sogenannten Kontrollnachrichten des Routingalgorithmus reduziert.

7.1.4 Geographische Verfahren

Geographische Verfahren (engl.: *geographic routing*, *gerouting*, *position based routing*) nutzen Informationen über die geographischen Positionen der einzelnen Knoten, um Wegewahlentscheidungen zu treffen. So kann mittels der Position der Senke beispielsweise der Knoten als weiterleitender Knoten ausgewählt werden, welcher der Senke am

nächsten ist (engl.: *greedy forwarding* oder *most forward within r*). Insbesondere ergeben sich bei Geographischen Verfahren neue Adressierungsmöglichkeiten in der Art, dass nun nicht mehr ein dedizierter Knoten sondern eine Region addressiert werden kann. Der dort befindliche Knoten wird dann als Senke identifiziert (engl.: *geocast*). Die Verwendung von Geographischen Verfahren setzt voraus, dass eine möglichst exakte Angabe über die Position der Knoten gemacht werden kann. Hier ist somit die Abhängigkeit von Positionsalgorithmen im Sensornetz gegeben.

Beispielhaft für Geographische Verfahren sei hier das *Greedy Perimeter Stateless Routing* (GPSR) [46] gennannt, das nach dem *greedy forwarding* die Senke erreicht. GPSR realisiert dabei zusätzlich eine *rechte-Hand-Regel*, mittels derer bei fehlenden Links lokale Minima in der Distanz umgangen werden.

7.1.5 Gossiping

Verfahren, die ohne jegliche Routinginformationen Daten von einer Quelle zur Senke leiten, werden von Karl et al. [45] unter sogenanntem *Gossiping* (d.: *Klatscherei*) subsummiert. Dabei ist die Grundidee, dass das Sammeln, Pflegen und Halten von Routinginformationen unter bestimmten Netz- und Applikationsparametern zu nicht zu rechtfertigenden Kosten (engl.: *overhead*) führt und daher nicht verwandt wird. Es wird quasi ohne Routinginformationen weitergeleitet.

Das einfachste Beispiel für ein solches Verfahren ist das bereits erwähnte Fluten von Nachrichten durch das Netz. Durch Erweiterungen wie beispielsweise Hinzufügen von probabilistischen oder zufälligen Elementen kann die Effizienz einen solchen Ansatzes verbessert werden. Ein Beispiel hierfür ist das *Gossip-Based Ad Hoc Routing* von Haas et al. [28].

7.1.6 Gradientenbasierte Verfahren

Gradientenbasierte Verfahren (engl.: *gradient-based*) legen eine Art „Kostenfeld“ über das Netz, welches bei der Senke minimal ist. Zu versendende Nachrichten werden dann entlang der sinkenden Kosten und somit dem negativen Gradienten des Kostenfeldes weitergeleitet.

Ein Beispiel für ein solches Verfahren ist der *Temporally-Ordered Routing Algorithm* (TORA) [81]. TORA flutet reaktiv, das heißt wenn zu versendende Daten vorliegen, ein sogenanntes *QUERY*-Paket durch das Netz. In diesem Paket ist die Adresse der Senke enthalten. Die Senke selbst oder Knoten, welche die Senke kennen, senden als Antwort ein sogenanntes *UPDATE*-Paket mit einer nach einer Metrik bestimmten Höhe oder Kosten. Die Empfänger eines *UPDATE*-Paketes erhöhen diesen Wert und speichern ihn für die gegebene Senke ab. Dadurch können die Empfänger gerichtete (engl.: *ordered*) Links zu der Senke auf das Netz legen. Pakete lassen sich nun entlang dieser Links durch das Netz ausliefern.

In *GRAdient Broadcast* (GRAB) [126] wird ein Szenario mit nur einer Senke zu Grunde gelegt. Die Senke sendet zunächst proaktiv eine sogenannte *advertisement* Nachricht aus. Diese wird durch das gesamte Netz geflutet. Basierend auf dieser Nachricht kann anhand einer Metrik (beispielsweise den Hops, die die Nachricht zurückgelegt hat) ein Kostenfeld aufgebaut werden, das an der Senke sein Minimum hat. Sendet eine Quelle ein Paket, so wird dies von den Knoten weitergeleitet, die gemäß dem Gradienten näher an der Senke sind. GRAB realisiert dadurch ein sogenanntes *multipath forwarding*, bei dem Nachrichten gegebenenfalls über mehrere Pfade in Richtung der Senke geleitet werden. Mit sogenannten *credits* kann man GRAB dazu zwingen, sogar Pfade mit zu benutzen, an denen der Gradient nicht streng monoton fällt. Über dieses *multipath forwarding* erhofft man sich eine größere Robustheit gegen Nachrichtenverluste.

7.2 Anforderung

In dem RFC (*Request for Comments*) 2501 *Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations* [10] ist festgelegt, welche Eigenschaften Wegewahlverfahren erfüllen müssen und anhand welcher Größen sich die Leistungsfähigkeit eines Verfahrens messen lässt. Die in dem RFC genannten Eigenschaften sind dabei: verteilte Funktionalität (engl.: *distributed operation*), Schleifenfreiheit (engl.: *loop freedom*), *demand-based operation*, *proactive operation*, Sicherheit (engl.: *security*), *sleep period operation* und *unidirectional link support*. Die Leistung des Routingverfahrens wird gemäß dem RFC gemessen anhand der Auslieferungsrate und Verzögerung (engl.: *End-to-end data throughput and delay*), *Route acquisition time*, *Percentage Out-of-Order Delivery* und der Effizienz (engl.: *Efficiency*). Die Effizienz wird dabei durch versandte Datenbits pro am Ziel ausgelieferte Datenbits bzw. Kontrollbits pro ausgelieferte Datenbits angegeben. Diese Größen müssen in Zusammenhang mit einem sogenannten *network context* erhoben werden. Dieser Kontext beschreibt das zu Grunde liegende Szenario bestehend aus Knotenzahl, Knotendichte, Linkkapazität, Anteil an unidirektionalen Links, Kommunikationsmustern, Mobilität und schlafenden Knoten.

Aus der oben aufgeführten Klassifizierung von Wegewahlalgorithmen lässt sich ableiten, dass bereits etliche verschiedene Wegewahlprotokolle existieren. Es wird jedoch ebenfalls deutlich, dass die Protokolle meist spezielle Annahmen an Netztopologien, Funkverhalten und zur Verfügung stehenden Informationen zu Grunde legen. Wie bereits beschrieben sind Geographische Verfahren von Positionsinformationen abhängig. Liegen diese Informationen nicht zu einer bestimmten Genauigkeit vor, können Geographische Verfahren nicht angewandt werden.

Broch et al. [5] geben in ihrer Arbeit einen ausführlichen simulativen Vergleich von DSDV, AODV, TORA und DSR (*Destination Sequence Routing*) [41][43][42] für Szenarien mit einem unterschiedlichen Grad an Mobilität und verschieden starkem Kommunikationsaufkommen. Die Ergebnisse zeigen ebenfalls deutlich, dass die Protokolle in verschiedenen Szenarien verschiedene Leistungen zeigen. So fällt die Auslieferungsrate

von DSDV gegenüber den anderen Verfahren bei hoher Mobilität drastisch ab. Dies ist auf das proaktive Verhalten von DSDV zurückzuführen. Ist die Mobilität im Vergleich zu der Akquirierung von Routinginformationen zu hoch, so versucht DSDV die Daten über nicht länger existierende Pfade zu leiten. Im Gegensatz dazu erzeugt beispielsweise TORA als reaktives Verfahren zusätzliche Wegewahlpakete und benötigt insbesondere in statischen Szenarien mehr Kontrollbits pro versandtem Datenbit als DSDV.

Die Arbeit von Broch et al. liefert einen Vergleich der verschiedenen Verfahren, jedoch geht aus der Arbeit nicht hervor, inwieweit Verluste auf dem Funkkanal und unidirektionale Links berücksichtigt wurden. Die in Kapitel 2.1.3 aufgeführten Erfahrungen durch Realisierungen von Sensornetzen zeigen, dass diese Effekte nicht vernachlässigt werden dürfen. Betrachtet man die Arbeitsweise der oben aufgeführten Verfahren DSDV und AODV, so erkennt man, dass diese in der Grundidee bidirektionale Links voraussetzen. Verfahren wie AODV besitzen daher spezielle Erweiterungen, um mit uniehaltendirektionalen Links umgehen zu können [82]. Dies führt jedoch zu komplizierten und fehlerträchtigen Implementierungen. Unter dieser Annahme und den von Broch et al. erzielten Ergebnissen in mobilen Szenarien erscheint es nicht sinnvoll, die Wegewahlentscheidungen auf bestimmten Links und Pfaden zu realisieren. Diese Informationen können bereits direkt nach ihrer Erhebung ungültig sein. Der weiterleitende Knoten adressiert in diesem Fall das Paket an einen vermeintlich direkt benachbarten Knoten, der nun aber nicht mehr erreichbar ist. Das Paket geht verloren und wird nicht ausgeliefert. Um die Praxistauglichkeit eines Wegewahlverfahrens unter allen Umständen gewährleisten zu können, muss ein anderer Ansatz gewählt werden.

Schliesst man Verfahren, die auf Link- und Pfadangaben basieren, und Geographische Verfahren aufgrund ihrer Abhängigkeit von Positionsinformationen aus, so bleibt die Möglichkeit der Gossiping Verfahren. Jedoch sollte die Entscheidung, welcher Knoten weiterleitet, nicht nach einem rein probabilistischen Prinzip zufällig bestimmt werden.

Der Ansatz von GRAB, bei dem ebenfalls die empfangenden Knoten die Weiterleitungsentscheidung treffen, dies jedoch in Abhängigkeit von einer Metrik zur Senke tun, erscheint vielversprechend. Dadurch, dass bei GRAB jedoch jeder Knoten in Richtung der Senke Pakete weiterleitet, erhält man lokal eine zu hohe Auslastung des Funkmediums. Dieses Problem wurde von Tse et al. [107] als sogenannter *Broadcast Storm* untersucht. GRAB skaliert somit nicht mit der Dichte des Netzes.

Aus den verwandten Arbeiten ergeben sich daher verschiedene Anforderungen an einen Wegewahlalgorithmus. So muss das Wegewahlverfahren link- und pfadunabhängig sein. Dadurch soll gewährleistet werden, dass das Verfahren unter verschiedensten physikalischen Einflüssen auf das Funkmedium weiterhin funktionsfähig bleibt. Auch sollen keine Einschränkungen an das Szenario oder die Topologie sowie die Kommunikationsmuster gemacht werden. Insbesondere soll das Verfahren die Eigenschaft des Funkmediums, dass ein Knoten die in seiner direkten Nachbarschaft versandten Nachrichten mithören kann, zu seinem Zweck ausnutzen. Dabei soll es, um schnell und fehlerfrei umgesetzt werden zu können, auf einfachen Regeln basieren.

7.3 Algorithmischer Ansatz

Der hier präsentierte *GRAPE*-Wegewahlalgorithmus kombiniert mehrere Ansätze. So legt bei *GRAPE* nicht der weiterleitende Knoten fest, welches der nächste Knoten des Pfades ist, sondern die empfangenden Knoten entscheiden, ob sie ein empfangenes Paket weiterleiten. Diese Entscheidung wird zum einen ähnlich wie bei GRAB auf Basis einer Metrik (im Folgenden Hops zur Senke) getroffen, die an der Senke ihr Minimum hat. Diese Metrik beschreibt dabei die lokale Distanz eines Knotens zur Senke. Zum anderen beobachten die empfangenden Knoten, ob das Paket bereits von einem Knoten näher zur Senke weitergeleitet wurde. So ist die Tatsache, dass ein empfangender Knoten sich näher an der Senke befindet, ein starker Indikator dafür, dass dieser das Paket weiterleitet. Das Paket folgt somit wie in Abbildung 7.1 dargestellt dem Gradienten der Metrik von der Quelle zur Senke.

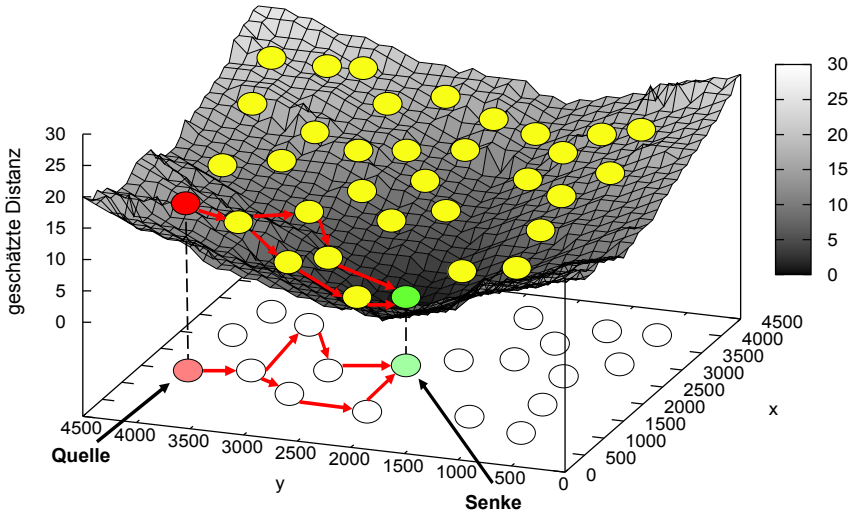


Abbildung 7.1: Gradient im Netz in Richtung einer Senke

Die Informationen über die Entfernungen zur Senke werden nicht aktiv akquiriert, sondern über passives Zuhören des Datenverkehrs sukzessive gelernt. So enthält jedes mit *GRAPE* versendete Datenpaket Informationen über die aktuelle Entfernung des Paketes von der Quelle als auch über die geschätzte Entfernung zur Senke. Die Auswertung dieser Informationen ermöglicht einem Knoten, Entfernungen zu verschiedenen Knoten im Laufe des Netzbetriebes zu erlangen. Die Senke leitet ein empfangenes für sie bestimmtes Paket weiter mit dem Minimalwert der verwendeten Metrik (hier: eine geschätzte Hop-Anzahl von 0) als geschätzte Entfernung zur Senke. Da kein anderer Knoten eine geringere Metrik zur Senke besitzt als die Senke selber, wird dieses Paket nicht weitergeleitet. Jedoch lernen die Knoten dadurch auch bei lediglich einseitigem Funkverkehr zwischen einer Quelle und einer Senke über die Entfernung zur Senke. So lernen die direkten Nachbarn der Senke durch das erneute Senden des Paketes, dass sie direkte

Nachbarn der Senke sind. Beim zweiten Paket zu der Senke erhalten die 2-hop-Nachbarn bereits die Information über die Distanz, bis nach n Paketen die Quelle in der Entfernung n die Distanz zur Senke kennt. Dieses Vorgehen wird im Folgenden *Backtracking* genannt.

Um ein simultanes Weiterleiten eines Paketes von allen Knoten näher zur Senke zu vermeiden, welches zu dem oben genannten *Broadcast Storm* Problem führen würde, verzögern die Knoten das Weiterleiten zufällig. Durch Beobachtung des Nachbarverhaltens kann ein potentieller weiterleitender Knoten feststellen, ob das Paket bereits erfolgreich weitergeleitet wurde. Somit bedarf es keines erneuten Sendens seinerseits. Durch dieses Verhalten wird die Anzahl der gesendeten Pakete reduziert und dem Broadcast Storm Problem entgegengewirkt. Dieses Vorgehen wird im Folgenden mit *Thinning* beschrieben, da die weiterzuleitenden Nachrichten ausgedünnt werden. Hierbei wird ebenfalls die Eigenschaft des Funkmediums ausgenutzt, dass jeder Knoten alle Nachrichten in seiner direkten Nachbarschaft hört.

Der *GRAPE*-Algorithmus leitet Pakete gegebenenfalls mittels des Konzepts des *Thinnings* über mehrere disjunkte Wege zur Senke. Um jedoch zu vermeiden, dass direkt beim ersten Senden der Quelle, bei dem das Paket noch nicht auf mehrere Wege repliziert wurde, das Paket durch Kollisionen verloren geht, wird das Paket gegebenenfalls mehrfach von der Quelle gesandt. Durch dieses *Source Rebroadcasting* sendet die Quelle daher eine Nachricht mehrfach, sofern die Quelle das Weiterleiten des Paketes durch einen benachbarten Knoten nicht registriert. Nach einer festgelegten maximalen Anzahl von erneutem Senden wird dieser Prozess abgebrochen. Das Versenden des Paketes wird als fehlgeschlagen angesehen.

Um praxistauglich zu sein, muss *GRAPE* auf Topologieänderungen, beispielsweise durch Mobilität reagieren können. Da *GRAPE* lediglich auf den Distanzen zur Senke und nicht auf exakten (und dabei jedoch möglicherweise falschen) Pfadinformationen arbeitet, muss die Distanz zur Senke bei Topologieänderungen angepasst werden. Im Falle von zu großen Distanzen werden die Pakete von der Quelle dennoch an die Senke ausgeliefert. Lediglich bei zu kurz geschätzten Distanzen kann der *GRAPE*-Algorithmus das Paket nicht erfolgreich ausliefern. Daher wird ein sogenanntes *Aging* der Distanzinformationen zu den Senken eingeführt. In einem Intervall (engl.: *aging interval*), das in Abhängigkeit von der Mobilität festgelegt wurde, erhöhen sich die geschätzten Distanzen zu den Senken auf jedem Knoten. Hier muss berücksichtigt werden, dass dies nicht simultan passiert. Da die Pakete entlang eines streng monoton fallenden Gradienten die Senke erreichen, dürfen im Gradienten keine Plateaus bzw. Sattelpunkte entstehen. Daher wachsen die Distanzinformationen immer um einen Wert (engl.: *aging step*) kleiner als die minimale Differenz in der gegebenen Metrik zwischen zwei Knoten. Im Falle der Hops steigt die Distanzinformation auf einem Knoten in den festgelegten Intervallen also beispielsweise um einen halben Hop. Somit kann auch bei asynchroner Erhöhung der Distanzen gewährleistet werden, dass der Gradient von der Quelle zur Senke streng monoton fallend bleibt.

7.4 Implementierung

Der *GRAPE*-Algorithmus verzichtet vollständig auf spezielle Routingpakete, wie *Route-Request* Pakete o.ä.. *GRAPE* zieht Informationen aus dem Datenverkehr und verfügt somit bei hohen Paketaufkommen über genauere Informationen, die ein effizientes Weiterleiten der Pakete ermöglichen. Um diese Informationen zu erhalten, wird jedes Datenpaket mit einem Paketkopf (engl.: *packet header*) versehen. Dieser Paketkopf enthält, wie in Tabelle 7.3 aufgeführt, eine Paket ID (*pid*), die Quelladresse (*src*), eine Zieladresse (*dst*) und den Vorgängerknoten (*fwd*), der das Paket gerade weitergeleitet hat. Zusätzlich wird der geschätzte Hopcount zu der Senke (*dst_hops*) sowie der aktuelle Hopcount des Paketes seit des Absendens an der Quelle (*src_hops*) angegeben.

pid	eindeutige Paket ID
src	Adresse der Quelle des Paketes
dst	Adresse des Zielknotens (der Senke)
fwd	Adresse des letzen weiterleitenden Knotens
dst_hops	geschätzte Hops zur Senke
src_hops	aktuell zurückgelegte Hops des Paketes

Tabelle 7.3: Paketkopf eines *GRAPE*-Datenpaketes

Das Verhalten von *GRAPE* beim Hören eines Datenpaketes basiert auf einfachen Regeln. So lässt sich das *GRAPE*-Wegewahlverfahren wie in Algorithmus 7.1 in wenigen Zeilen darstellen. Aus jedem Paket werden zunächst die Informationen über das Netz aus dem Paketkopf *ph* gewonnen und in eine *Weiterleitungstabelle* ft (engl.: *forwarding table*) eingetragen (Zeile 4-10). Erkennt man, dass der Hopcount zu diesen Knoten geringer ist als bisher angenommen, werden diese Informationen übernommen. So wird die Distanz zum Knoten, der das Paket weitergeleitet hat (*fwd*), auf 1 Hop gesetzt, da dieser offensichtlich ein direkter Nachbar des empfangenden Knotens ist (Zeile 4). Ist die Anzahl der bisher angenommen Hops zur Quelle größer oder gleich der Anzahl der Hops, die das Paket von der Quelle *src* bis zum empfangenden Knoten zurückgelegt hat, so wird die neue Hopanzahl in der Weiterleitungstabelle für die Quelle gespeichert (Zeile 5-7). Sind die geschätzten Hops zur Senke in der Weiterleitungstabelle größer als die vom direkten Nachbarn *fwd* geschätzen Hops zur Senke *dst* + 1, so wird der um eins erhöhte Wert in die Weiterleitungstabelle eingetragen (Zeile 8-10). Die Erhöhung um eins resultiert aus der Annahme, dass man im Zweifelsfalle weiter als der direkte Nachbar von der Quelle entfernt ist. Werden die Hops als Metrik zu Grunde gelegt, so bedeutet dies, dass man genau einen Hop weiter entfernt ist als der direkte Nachbar.

Der beschriebene Teil des Algorithmus ermöglicht es, auf spezielle Routingpakete zu verzichten. Hier wird die Eigenschaft des Funkmediums genutzt, dass eine Nachricht auf dem Medium von allen umliegenden Knoten gehört werden kann. Dadurch gewinnt jeder hörende Knoten bei jedem Paket Informationen über die Netztopologie.

Algorithmus 7.1 *GRAPE*-Algorithmus

```
1: while TRUE do                                                    // läuft periodisch
2:     empfange Paket p und interpretiere Paketkopf ph
3:                                  // aktualisiere Einträge in Weiterleitungstabelle ft[]
4:     ft[ph.fwd].hops = 1                                          // für den Vorgängerknoten
5:     if ft[ph.src].hops >= ph.src_hops then
6:         ft[ph.src].hops = ph.src_hops          // aktualisiere den Eintrag für die Quelle in ph
7:     end if
8:     if ft[ph.dst].hops >= ph.dst_hops + 1 then
9:         ft[ph.dst].hops = ph.dst_hops + 1      // aktualisiere den Eintrag für die Senke in ph
10:    end if
11:      // leite weiter, wenn die Distanz kleiner oder „unbekannt“ als Distanz in ph angegeben ist
12:    if ph.dst_hops = UNKNOWN_HOP_COUNT or ft[ph.dst].hops < ph.dst_hops then
13:        ph.src_hops + +
14:        ph.fwd = this
15:        ph.dst_hops = ft[ph.dst].hops
16:        forward p
17:    end if
18: end while
```

Nachdem die Informationen aus dem Paketkopf des Datenpaketes gewonnen wurden, wird die Weiterleitungsentscheidung getroffen. Ist die geschätzte Distanz (die geschätzte Hopanzahl) zur Senke *dst* geringer als diejenige, die im Paketkopf angegeben wird, oder entspricht die angegebene Hopanzahl im Paket der maximalen Hopanzahl (*UNKNOWN_HOP_COUNT*), was im initialen Fall bei der ersten Nachricht der Fall wäre, so wird das Paket weitergeleitet (Zeile 12-17). Es ist zu beachten, dass die Verzögerung und das Konzept des *Thinnings* hier in der Weiterleitung gekapselt sind (Zeile 16) und nicht gesondert aufgeführt werden. Vor der Weiterleitung des Paketes passt *GRAPE* jedoch den Paketkopf des Datenpaketes an. So wird die aktuelle Anzahl der Hops, die das Paket zurückgelegt hat, um eins erhöht (Zeile 13) und der weiterleitende Knoten *fwd* wird auf diesen Knoten gesetzt (Zeile 14). Zudem wird die geschätzte Hopzahl zur Senke *dst* in dem Paketkopf auf den Wert aus der Weiterleitungstabelle *ft* des Knotens gesetzt. Somit kann der empfangende Knoten aus den Informationen des Paketkopfes auf gleiche Weise wieder Topologieinformationen über das Netz gewinnen.

7.5 Evaluation

Um die Verhaltensweise des *GRAPE*-Algorithmus zu verstehen und von anderen Verfahren abzugrenzen, wurde zunächst eine simulative Untersuchung durchgeführt. Dadurch können einzelne Charakteristika des Algorithmus einfacher in speziell gewählten Szenarien verdeutlicht werden. Im Hinblick auf die Praxistauglichkeit hat der Autor Experimente mit den *Pacemate*-Sensorknoten durchgeführt. Dabei ist *GRAPE* in den

Messungen auf die Auslieferungsrate, die Verzögerung und die Effizienz, die alle in dem bereits erwähnten RFC 2501 aufgeführt sind, untersucht worden.

7.5.1 Simulative Evaluation

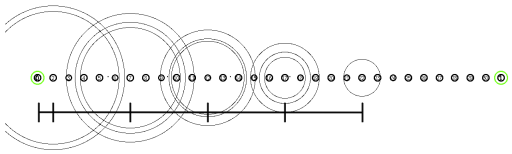
Im Rahmen der simulativen Untersuchung wurden zunächst zwei Szenarien überprüft, die die Stärken des *GRAPE*-Wegewahlverfahrens verdeutlichen. In einer dritten Simulation werden die verschiedenen im RFC 2501 aufgeführten Größen zur Evaluation von Routingverfahren gemessen und verglichen. Zur Simulation wurde bei allen Messungen der bereits genannte Netzsimulator *ns2* in der Version 2.29 verwendet.

Optimale Wege

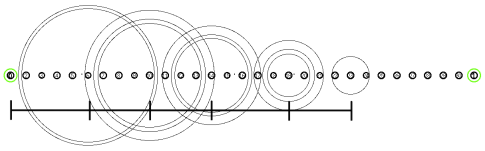
Bereits in statischen Szenarien verschwinden und entstehen permanent Links zwischen einzelnen Knoten, was dazu führt, dass Pfade, die durch ein pfadbasiertes Routingverfahren gewählt wurden, schnell nicht mehr optimal oder gar ungültig sind. Pfadbasierte Protokolle wie AODV bestimmen den Pfad zunächst beispielsweise über *Route Request*-Pakete. Während des Flutens solcher Nachrichten durch das Netz treten durch den bereits erwähnten *Broadcast Storm* vermehrt Kollisionen auf dem Funkmedium auf und *Route Request*-Pakete gehen verloren. Unter anderem dadurch kann es beispielsweise in AODV auftreten, dass nicht der optimale (kürzeste) Pfad gefunden wird. Ist jedoch einmal ein Pfad gefunden, wird dieser verwendet, bis der Pfad nicht mehr gültig ist und ein neuer *Route Request*-Prozess angestoßen wird.

Wie beschrieben nutzt der *GRAPE*-Algorithmus einen Gradienten von der Quelle zur Senke, der durch den Nachrichtenverkehr mittels des mitgesendeten Paketkopfes entsteht. Im initialen Zustand ist die Distanz zwischen Quelle und Senke unbekannt. Gemäß der aufgeführten Implementierung wird ein Paket, das als geschätzte Entfernung den Maximalwert *UNKNOWN_HOP_COUNT* hat, weitergeleitet. Leiten Knoten, die eine Entfernung zur Senke kennen, das Paket weiter, so gewinnen alle umliegenden Knoten nach dem oben beschriebenen Verfahren durch die Interpretation des Paketkopfes die Information über die Distanz zur Senke. Ist der Gradient komplett aufgebaut, so leitet nur der Knoten weiter, der wirklich näher im Bezug auf die verwendete Metrik (beispielsweise: Hops) an der Senke ist. Der pfadunabhängige *GRAPE*-Algorithmus wählt daher immer die optimalen Wege zur Senke.

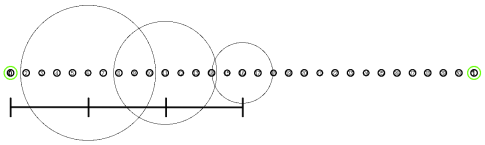
Abbildung 7.2 zeigt das Ergebnis einer Simulation auf einem Szenario, welches das Nutzen des optimalen Weges von *GRAPE* im Vergleich zu anderen pfadbasierten Wegewahlverfahren verdeutlicht. Die Knoten sind auf einer Linie im 50 m Abstand angeordnet bei einer Kommunikationsreichweite von 250 m. Die Quelle ist dabei der Knoten ganz am linken Ende der Reihe und die Senke der Knoten am rechten Ende. Die Nachrichten wurden zunächst mit dem Wegewahlverfahren DSR und anschließend mit dem Wegewahlverfahren AODV versandt. Die Implementationen der Verfahren sind in der *ns2*-Distribution enthalten. Abschließend wurden die Nachrichten mit dem *GRAPE*-Algorithmus versandt



(a) DSR



(b) AODV



(c) *GRAPE*

Abbildung 7.2: Die von den Verfahren gewählten weiterleitenden Knoten in einem geradlinigen Szenario

(Abbildung 7.2(c)). Die Ringe zeigen dabei, wie sich Pakete auf dem Funkmedium ausbreiten. Der Radius der Ringe beschreibt dabei die räumliche Ausbreitung der Funkwelle. Im Zentrum der Ringe ist dabei der Sender (Weiterleiter) eines Pakets. Bei der gegebenen Funkreichweite von exakt 250 m und einer Distanz von 50 m zwischen den Knoten sollte im optimalen Fall jeder fünfte Knoten das Paket weiterleiten, um den kürzesten Weg zur Senke am rechten Ende der Reihe zu erhalten. Man sieht deutlich, dass die vorliegende Implementierung von DSR den direkten Nachbarn der Quelle zum Weiterleiten des Pakets wählt (Abbildung 7.2(a)), was in dem gegebenen Szenario nicht sinnvoll erscheint. Abbildung 7.2(b) zeigt, dass AODV bei der zweiten und dritten Weiterleitung lediglich 4 Knoten überbrückt. Durch die permanente Anpassung des Gradienten bei jedem passierenden Datenpaket erreicht *GRAPE* den kürzesten Weg, so dass jeder fünfte Knoten das Paket weiterleitet.

Disjunkte Wege

Die Erfahrungen aus den praktischen Umsetzungen von Sensornetzen, die in Kapitel 2.1.3 dargestellt wurden, und aus den eigenen Umsetzungen [63] zeigen, dass Links und somit Pfade unter realen Umständen und Einflüssen sehr instabil sind. In dichten Netzen, in denen mehrere Pfade von der Quelle zur Senke existieren, werden Ansätze verfolgt, Pakete über verschiedene disjunkte Pfade zur Quelle parallel zu versenden (engl.: *multipath routing*). Dabei werden neue Verfahren konzipiert wie in [108] oder existierende Verfah-

ren erweitert. So existiert mit AODV-BR [53] eine Multipath-Erweiterung zu AODV und mit [76] eine Erweiterung zu DSR. Ziel der Verfahren ist es, durch die Nutzung mehrerer Pfade eine höhere Fehlertoleranz bei der Existenz von ungültigen Pfaden zu erreichen und somit auch bei geringer Linkstabilität hohe Auslieferungsraten zu erreichen.

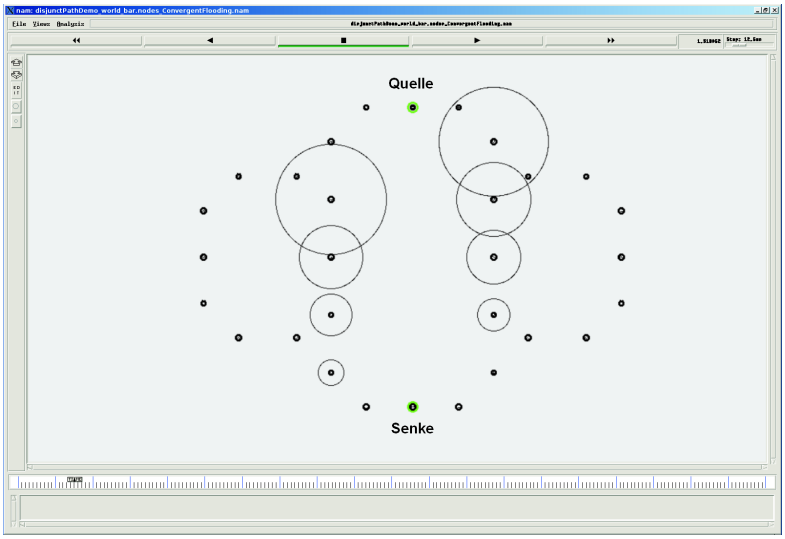


Abbildung 7.3: Disjunkte kürzeste Wege im *GRAPE*-Algorithmus

Der *GRAPE*-Wegewahlalgorithmus nutzt inhärent mehrere mögliche Wege zur Senke, da jeder Knoten die Pakete entlang des streng monoton fallenden Gradienten weiterleitet. Abbildung 7.3 zeigt, wie ein Paket nach Aufbau des Gradienten auf beiden möglichen kürzesten Wegen von der Quelle zur Senke weitergeleitet wird. In einem dichten Netz werden die disjunkten Pfade mittels des zuvor aufgeführten Konzept des *Thinnings* erzwungen. Dabei leiten nur die Knoten Pakete entlang des streng monoton fallenden Gradienten weiter, die keinen anderen weiterleitenden Knoten hören, der näher an der Senke ist. Liegen ein weiterleitender Knoten n jedoch so weit von einem anderen weiterleitenden Knoten m entfernt, dass n das versandte Paket von m nicht empfängt, so leitet n ebenfalls das Paket weiter. Das Paket wird somit auf zwei disjunkten Wegen weitergeleitet.

Auslieferungsrate, Verzögerung und Effizienz

Neben den Eigenschaften wie die Nutzung optimaler Wege und mehrerer disjunkter Pfade ist in der dritten simulativen Untersuchung die Leistungsfähigkeit des *GRAPE*-Wegewahlverfahrens von Interesse. Hierbei werden die Auslieferungsrate (engl.: *delivery ratio*), die Verzögerung (engl.: *end-to-end delay*) und die Effizienz (die Effizienz als die gesendeten Bytes pro ausgelieferte Bytes) untersucht. Der *GRAPE*-Algorithmus wurde dabei mit den in der *ns2*-Distribution enthaltenen Wegewahlverfahren AODV und

DSDV verglichen. Um die Ergebnisse der Simulation in einen Zusammenhang zu existierenden Arbeiten stellen zu können, wurde für die Evaluation das von Broch et al. bei ihrer Untersuchung von AODV, DSDV, DSR und TORA verwendete Szenario genutzt. Dabei wurde zur Kenntnis genommen, dass das zur Erzeugung der zufälligen Bewegung der Knoten in der Simulationsfläche verwendete *Random Waypoint* Bewegungsmodell, bei dem die Knoten sich immer wieder zu zufälligen Zielpunkten auf der Fläche bewegen, spezielle Charakteristika aufweist [128]. Auf diese wird an dieser Stelle jedoch nicht weiter eingegangen. Broch et al. haben die verschiedenen Wegewahlverfahren mit verschiedenen Graden an Mobilität evaluiert. Dies bedeutet, dass die Standzeiten der Knoten, bevor sie einen neuen Zielpunkt wählen (engl.: *pause times*), sowie die maximalen Geschwindigkeiten der Knoten innerhalb des Bewegungsmodells variiert wurden. In der hier durchgeführten Evaluation wurde das Szenario von Broch et al. mit der höchsten Mobilität gewählt. Dabei ist die Standzeit der Knoten nach dem Erreichen eines Zielpunktes gleich 0 und die Geschwindigkeiten werden zwischen 0 und 20 m/s zufällig gewählt. Als Funkübertragungsmodell verwenden Broch et al. das *two-ray-ground-reflection-Modell* [54]. Dieses Modell besagt, dass bei der Funkübertragung die Reflektion des vom Sender ausgesandten Signals in Abhängigkeit von der Verzögerung den Empfang beeinflusst. Bei Phasengleichheit kann die Reflektion vom Boden das Signal beim Empfänger verstärken. Liegt jedoch eine Verschiebung in der Phase vor, so kann dass Signal beim Empfänger abgeschwächt oder ausgelöscht werden. Eine Übersicht über die gesamten Simulationsparameter gibt Tabelle 7.4. Insgesamt wurden die Simulationen zehnmal mit verschiedenen zufälligen Bewegungsmustern durchgeführt. Die Parameter des *GRAPE*-Algorithmus sind in Tabelle 7.5 aufgeführt. Dabei wird das Intervall, in dem sich der geschätzte Hopcount zu allen Quellen um einen halben Hop für das sogenannte *Aging* erhöht, auf 0,25 s gesetzt. Somit ist es *GRAPE* möglich, auch im mobilen Szenario die Topologieänderungen zu registrieren.

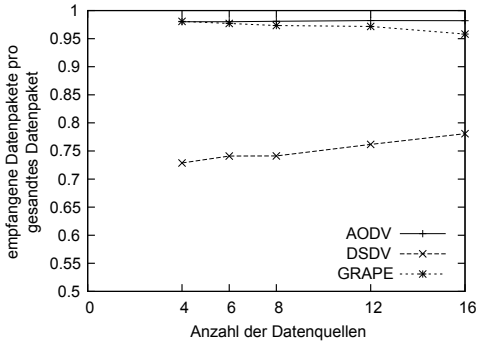


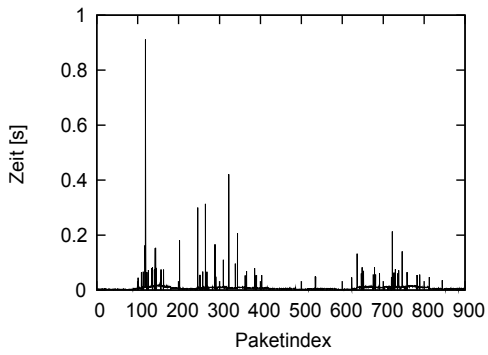
Abbildung 7.4: Auslieferungsrate von *GRAPE* im Vergleich zu AODV und DSDV

Broch et al. zeigen in ihrer Arbeit, dass die von ihnen untersuchten Wegewahlalgorithmen bei steigender Mobilität starke Abnahmen in der Auslieferungsrate der Pakete aufweisen. Abbildung 7.4 zeigt, dass *GRAPE* in einem hochmobilen Szenario 95 % der Pakete ausliefert. Durch das ständige Interpretieren des Datenverkehrs und dem darauf basie-

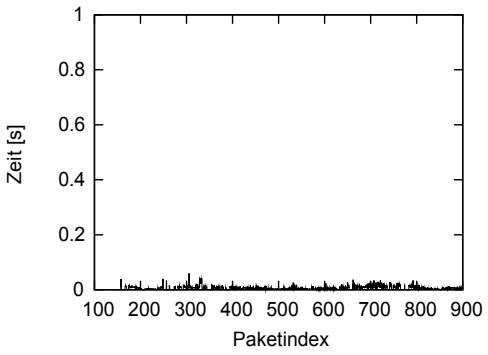
renden Anpassen des Gradienten kann *GRAPE* sich schnell an die Topologieänderungen anpassen. Im Vergleich dazu ereicht DSDV im gleichen Szenario gerade eine Auslieferungsrate von 75 %. Dies lässt sich auf veraltete Einträge in den von DSDV proaktiv erstellten Routingtabellen zurückführen.

Neben der Auslieferungsrate ist die Verzögerung bei der Auslieferung eine wichtige Größe bei der Evaluation von Wegewahlverfahren. Insbesondere für Anwendungen mit zeitkritischen Messungen ist es von großer Wichtigkeit, dass registrierte Daten umgehend von der Quelle zur Senke übertragen werden, um zeitnah entsprechende Aktionen einzuleiten. Abbildung 7.5 zeigt beispielhaft die durch die einzelnen Verfahren erzeugten Verzögerungen während eines Simulationsdurchlaufes mit vier Datenquellen. In Abbildung 7.5(a) ist deutlich das reaktive Verhalten des AODV-Algorithmus zu erkennen. AODV erzeugt lange Verzögerungen in der Auslieferung von bis zu einer Sekunde, welche durch die durchzuführenden Pfadakquirierungen hervorgerufen werden. So muss AODV in einem hochmobilen Szenario zunächst eine *Route Request* durch das Netz fluten und auf einen *Route Reply* warten, bevor die eigentlichen Daten versandt werden können. Das DSDV Wegewahlverfahren erzeugt wesentlich weniger Verzögerungen, wie in Abbildung 7.5(b) dargestellt, da hier die Pfade proaktiv in gleichbleibenden Intervallen akquiriert werden. Hierbei ist jedoch in Betracht zu ziehen, dass die Auslieferungsrate von DSDV für vier Datenquellen zwischen 70 % und 75 % liegt. In Abbildung 7.5(c) wird deutlich, dass der *GRAPE*-Wegewahlalgorithmus trotz ständiger Änderungen in der Topologie nur geringe Verzögerungen produziert. Da *GRAPE* sich nicht auf Pfade festlegt, müssen die weiterleitenden Knoten bei Topologieänderungen nicht explizit bestimmt werden, sondern ergeben sich automatisch aus dem Datenverkehr.

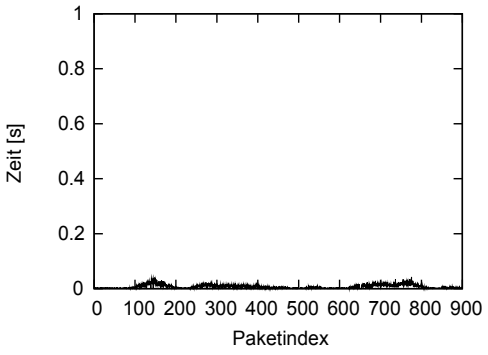
Die dritte untersuchte Größe bezieht sich auf die Effizienz des Wegewahlverfahrens. Hier werden die gesandten Bytes in das Verhältnis zu den ausgelieferten Bytes gesetzt. Dies erfasst, wieviel Aufwand (in Form von gesandten Nachrichten) ein Wegewahlalgorithmus betreibt, um ein Paket von der Quelle zur Senke zu versenden. Abbildung 7.6 zeigt dabei den Quotienten aus versandten Bytes und ausgelieferten Bytes für die drei Wegewahlverfahren für verschiedene Anzahlen von Datenquellen. Es ist erkennbar, dass *GRAPE* für jedes ausgelieferte Byte neun Bytes gesandt hat, unabhängig von der Anzahl der Datenquellen und des dadurch entstehenden Datenverkehrs. Die kürzesten Pfade zwischen Quelle und Senke sind in dem gegebenen Szenario im Durchschnitt 2,3 Hops. Dies bedeutet, dass der *GRAPE*-Wegewahlalgorithmus in dem gegebenen Szenario um einen Faktor vier schlechter ist als ein optimales Verfahren, welches ohne Informationen den kürzesten Weg zwischen Quelle und Senke nutzt. AODV ist dagegen lediglich um einen Faktor 1,5 schlechter als ein optimales Verfahren. Das vermeintlich schlechte Ergebnis von *GRAPE* ist darin begründet, dass der Algorithmus mehrere Wege zwischen Quelle und Senke nutzt. Dadurch werden Daten redundant versandt, was zu einer höheren Anzahl von gesandten Bytes pro ausgeliefertes Byte führt.



(a) AODV



(b) DSDV



(c) *GRAPE*

Abbildung 7.5: Verzögerungen bei der Paketauslieferung durch (a) AODV, (b) DSDV und (c) *GRAPE* im gleichen Szenario

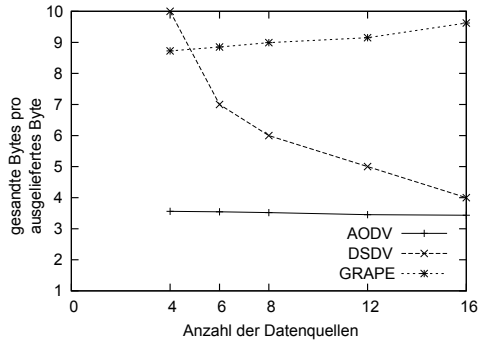


Abbildung 7.6: Versandte Bytes pro ausgelieferte Bytes von *GRAPE* im Vergleich zu AODV und DSDV

7.5.2 Evaluation auf Pacemate-Sensorknoten

Eine Evaluation auf Sensorknoten in der Praxis ist notwendig, um die Aussage der Ergebnisse der obigen Simulationen bewerten und die Praxistauglichkeit des *GRAPE*-Algorithmus bestimmen zu können.

Aufbau und Parameter

Für das Experiment wurde das in der Simulation verwandte Szenario, welches von Broch et al. verwandt wurde, nachgestellt. Dabei wurden die Dimensionen des Szenarios an die Eigenschaften der *Pacemate*-Sensorknoten angepasst. Die Messparameter sind in Tabelle 7.6 dargestellt. Legt man den *Pacemate*-Sensorknoten eine Funkreichweite von 3 m zu Grunde, wenn sie auf dem Boden im Freifeld liegen, so ergibt sich eine Simulationsfläche von 18 m × 4 m. Die Knoten wurden nicht bewegt, sondern statisch platziert. Die Anzahl der Datenquellen wurde auf vier beschränkt. Außerdem wurde die Simulationszeit halbiert auf 450 s.

Auf den 50 ausgebrachten Knoten befand sich das in Kapitel 4 eingeführte *Surfer OS*-Betriebssystem. Ein Dienst, der das *GRAPE*-Wegewahlverfahren realisiert, wurde auf die Knoten migriert. Außerden wurde der Funkprotokollstapel als Dienst durch einen neuen Protokollstapel ersetzt, der den *GRAPE*-Algorithmus einbindet. Zusätzlich wurden weitere Dienste auf die Knoten migriert, welche die Statistiken der Routingverfahren aufzeichnen und ein Auslesen der Werte ermöglichen, sowie ein Dienst, der Pakete gemäß den Simulationsparametern versendet. In mehreren Durchgängen, bei denen die Statistiken sowie die Informationen des *GRAPE*-Algorithmus zurückgesetzt wurden, sind zufällig verschiedene Knoten als Quellen und Senken ausgewählt worden. Mit einem Zeitversatz von wenigen Sekunden wurden die Übertragungen gestartet.

Messergebnisse

Die von allen Knoten aufgezeichneten Statistiken wurden ausgelesen und sind in Tabelle 7.7 zusammengefasst. Die in dem Szenario durch den *GRAPE*-Algorithmus ausgelieferten Pakete haben im Schnitt 3,8 Hops von der Quelle zur Senke benötigt. Damit entspricht das Szenario im Hinblick auf die Knotendichte näherungsweise dem zuvor simulativ untersuchten Szenario, bei dem die durchschnittliche Pfadlänge 2,3 Hops betrug.

Im Vergleich zu den simulativen Untersuchungen des *GRAPE*-Algorithmus ist die Auslieferungsrate bei der Messung auf den *Pacemate*-Sensorknoten mit 42,2 % deutlich niedriger. Gleichzeit ist die Effizienz mit 36,7 versandten Paketen pro ausgeliefertem Paket deutlich schlechter als in der Simulation. Ein Grund für das schlechte Verhältnis zwischen versandten und ausgelieferten Paketen liegt dabei in der geringen Anzahl ausgelieferter Pakete.

Das Ergebnis zeigt, dass die mit dem Netzwerksimulator erzielten Ergebnisse nicht mit denen auf *Pacemate*-Sensorknoten übereinstimmen. Daraus lässt sich ableiten, dass die

Simulationsfläche	1500 m × 300 m
Anzahl der Knoten	50
Kommunikationsradius	250 m
Funkübertragungsmodell	Two-ray-ground-reflection-Modell Höhe der Antenne 1,5 m
Bewegungsmodell	Random Waypoint Geschwindigkeit zwischen 0 und 20 m/s *pausetime* 0 s
Datenquellen	4, 6, 8, 12, 16
Sendeintervall	4 Pakete pro Sekunde (für jede Quelle)
Paketgröße	64 byte
Simulationszeit	900 s

Tabelle 7.4: Parameter der *ns2*-Simulation

aging interval	0,25 s
aging step	0,5
max. Sendewiederholungen beim ersten Hop	3

Tabelle 7.5: Verwendete Parameter des *GRAPE*-Algorithmus

Simulationsfläche	18 m × 4 m
Anzahl der Knoten	50 *Pacemate* Sensorknoten
Kommunikationsradius	ca. 3 m (auf dem Boden liegend)
Bewegungsmodell	statisch
Datenquellen	4
Sendeintervall	4 Pakete pro Sekunde (für jede Quelle)
Paketgröße	64 byte
Simulationszeit	450 s

Tabelle 7.6: Parameter der *GRAPE* Freifeldmessung

	Auslieferungsrate	Effizienz	durchschnittliche Distanz (Hops)
1. Messung	18,1 %	80,5	5,2
2. Messung	46,7 %	26,2	4,1
3. Messung	40,3 %	29	3,9
4. Messung	63,6 %	11,2	1,9
Durchschnitt	42,2 %	36,7	3,8

Tabelle 7.7: Ergebnisse der Evaluation des *GRAPE*-Algorithmus auf 50 *Pacemate*-Sensorknoten

in der Simulation verwendeten Modelle wie das Funkübertragungsmodell die realen Gegebenheiten nicht vollständig abbilden. Insbesondere wird deutlich, dass das simulierte Verhalten und die Leistungsfähigkeit eines Algorithmuses stark von diesen Modellen abhängen.

7.6 Ergebnis

In diesem Kapitel wurde das neue Wegewahlverfahren *GRAPE* für Funknetze entwickelt. Dazu sind zunächst existierende Wegewahlverfahren klassifiziert worden. Neben der Unterteilung in proaktive und reaktive Verfahren, die festlegt, wann Informationen über den Weg zur Senke gesammelt werden, wurden die Verfahren ebenfalls nach der Art der gesammelten Informationen unterschieden. Die Wegewahlalgorithmen sind in Distanzvektor-, Link-State-, Geographische und Gradientenbasierte Verfahren unterteilt worden. Dabei wurde deutlich, dass viele Verfahren spezielle Anforderungen wie bidirektionale Links an die zu Grunde liegende Netztopologie stellen. Um die Verfahren auch in anderen Topologien zum Einsatz zu bringen, sind meist aufwendige Erweiterungen nötig.

Unter der Vorgabe, dass ein Wegewahlverfahren keine Einschränkungen an die Netztopologie stellen soll, ist in diesem Kapitel der *GRAPE*-Algorithmus entworfen worden. Der *GRAPE*-Algorithmus basiert dabei nicht auf Links zwischen einzelnen Knoten sondern auf einem netzweiten Gradienten zur Senke. Im Gegensatz zu Distanzvektor- und Link-State-Verfahren entscheidet in *GRAPE* nicht der sendende Knoten, welcher Nachbarknoten die Nachricht weiterleitet. Die empfangenden Knoten treffen eigenständig die Weiterleitungsentscheidung. Die Entscheidung basiert dabei auf Informationen, die nicht aktiv akquiriert werden, sondern passiv durch Zuhören auf dem Funkmedium gesammelt werden. Durch dieses Lauschen sammelt jeder Knoten Informationen über die Topologie des Netzes und die Distanzen zu anderen Knoten. Außerdem erkennt ein Knoten, ob ein Paket bereits erfolgreich von anderen Knoten in Richtung der Senke weitergeleitet wurde.

Der *GRAPE*-Algorithmus basiert auf der einfachen lokalen Entscheidung jedes Knotens, ob er näher an der Senke ist als der Sender des Paketes. Durch das Sammeln der Topologieinformationen durch Lauschen auf dem Funkkanal verzichtet *GRAPE* auf spezielle Pakete zur Aufrechterhaltung der Topologieinformation. Hierdurch lässt sich der *GRAPE*-Algorithmus in wenigen Zeilen Pseudocode spezifizieren und somit einfach umsetzen.

Die simulative Untersuchung hat gezeigt, dass der *GRAPE*-Algorithmus die im Sinne der zu Grunde liegenden Metrik optimalen Wege nutzt. Weiterhin werden Pakete über mehrere disjunkte Wege von der Quelle zur Senke geleitet, sofern dies die Netztopologie zulässt, was die Robustheit des Algorithmus gegen instabile Links vergrößert. Im Vergleich mit ADOV und DSDV erreicht *GRAPE* in den untersuchten mobilen Szenarien eine Auslieferungsrate von 95 %. Gleichzeitig erzeugt *GRAPE* im Vergleich zu AODV wesentlich geringere Verzögerungen bei der Auslieferung der Pakete. Dies liegt darin

begründet, dass *GRAPE* keinen gesonderten Prozess zur Akquirierung von Pfadinformationen vor dem Versenden der eigentlichen Daten beginnt, sondern die Informationen permanent aus dem Datenverkehr gewinnt.

In einer Untersuchung des *GRAPE*-Algorithmus auf *Pacemate*-Sensorknoten wurde das simulativ untersuchte Szenario nachgestellt. Die Ergebnisse der Untersuchung zeigen jedoch, dass der *GRAPE*-Algorithmus lediglich 42,2 % der Pakete ausliefert. Dies bedeutet, dass der *GRAPE*-Algorithmus in dem gewählten Szenario nicht die gewünschten Anforderungen an die Robustheit erfüllt. Hierdurch wird zum einen die Notwendigkeit von experimentellen Untersuchungen auf Sensorknoten deutlich. Zum anderen ist erkennbar, dass das Ersetzen von Diensten wie dem Wegewahlalgorithmus vor Ort von Vorteil ist, um das optimale Verfahren zu verwenden. Mit Hilfe der *Pacemate*-Sensorknoten und des *Surfer OS*-Betriebssystems können Dienste ersetzt und so während der Experimente die Wegewahlverfahren auf den Knoten ausgetauscht werden, was die Durchführung verschiedener Messungen beschleunigt.

8 Entwicklerorientiertes Anwendungsdesign mittels SOA in WSNs

Das Ziel dieser Arbeit ist es, dem Anwender der Sensornetztechnologie eine Abstraktion und Werkzeuge zur Verfügung zu stellen, um ohne Expertenwissen aus den Bereichen der verteilten Systeme und Sensornetze eigene Anwendungen erstellen und pflegen zu können. Aus den in Kapitel 2.1.3 aufgeführten Beispielen technischer Realisierungen ist ersichtlich, dass dies mit dem aktuellen Stand der Technik nicht möglich ist. Es bedarf spezieller Analysen des Anwendungsszenarios unter anderem mit Blick auf Datenaufkommen und Netzdichte sowie der Untersuchung der Funkcharakteristika am Einsatzort. Neben dem aufwendigen Prozess der Analyse des Applikationsszenarios ist auch das Expertenwissen erforderlich, um die gewonnenen Daten zu interpretieren und basierend auf den Ergebnissen Designentscheidungen beispielsweise über die zu verwendenden Algorithmen für die Applikation zu treffen.

Der aktuelle Prozess zur Entwicklung von Sensornetzanwendungen kann daher wie in Abbildung 8.1 dargestellt werden. Ein potentieller Anwender von Sensornetztechnologie formuliert seine Problemstellung und Anforderungen, wie beispielsweise die Überwachung von Ereignissen oder die Verfolgung von Objekten. Diese Beschreibung übergibt der Anwender an einen Sensornetzexperten, der das Anwendungsszenario modelliert und analysiert. Unter Berücksichtigung der möglichen Umgebungseinflüsse wählt der Experte verschiedene Algorithmen und erstellt die Sensornetzanwendung. Während des erfolgreichen Einsatzes des Sensornetzes erhält der Anwender Daten aus dem Sensornetz, welche neue Erkenntnisse bringen und gegebenenfalls neue Fragestellungen aufwerfen, woraus sich somit weitere neue Anforderungen an die Sensornetzapplikation ergeben. Diese Anforderungen werden erneut dem Experten vorgelegt. Somit ist es dem Anwender nicht möglich, eigenständig Änderungen an der Anwendung vorzunehmen. Wie bereits erwähnt lassen sich hier Parallelen zur Nutzung der Informationstechologie in den frühen 1960er Jahren ziehen, denn auch damals hat die Nutzung und Bedienung eines Rechners spezielle Fachkenntnisse erfordert.

In diesem Kapitel wird beschrieben, wie die in den vorhergehenden Kapiteln erarbeiteten Ergebnisse für das Ziel der schnellen und einfachen Anwendungsentwicklung und -pflege für Sensornetze genutzt werden können. Dabei steht insbesondere der Anwender der Sensornetztechnologie im Fokus. Im Folgenden werden zunächst die Anforderungen formuliert. Darauffolgend wird die Architektur einer umgesetzten Infrastruktur zur Anwendungsentwicklung und -pflege beschrieben. Es wird demonstriert, wie eine Applikation unter Nutzung der Infrastruktur erstellt und modifiziert werden kann. Abschließend wird

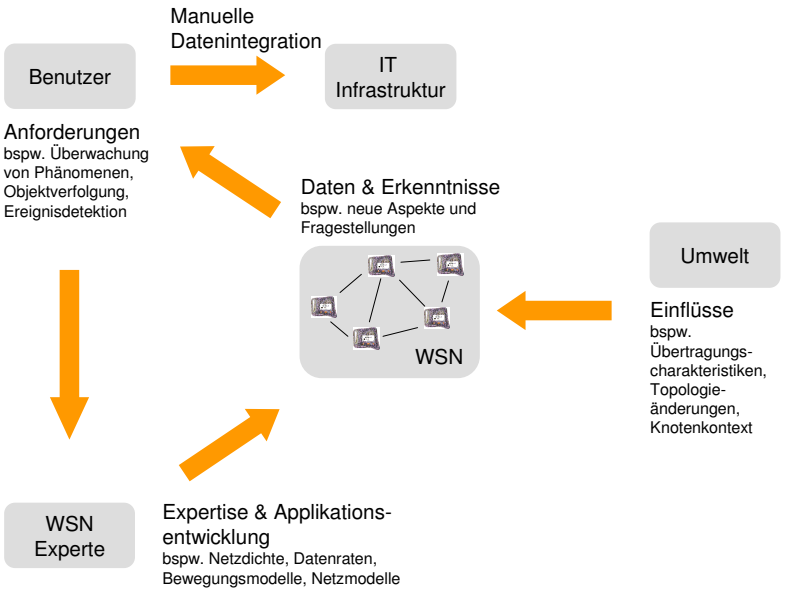


Abbildung 8.1: Aktueller WSN Applikationsentwicklungszyklus

in Form eines Ausblicks beschrieben, wie die Applikationsentwicklung weiterentwickelt werden kann.

8.1 Anforderungen

Durch das in Kapitel 4 vorgestellte service-orientierte Betriebssystem *Surfer OS* wurde das Paradigma der Service-Orientierung auf den Sensorknoten realisiert. Durch diese Abstraktion ist es möglich, einzelne Funktionalitäten in einzelnen Diensten zu kapseln, welche nur bei Bedarf genutzt werden. Dem Nutzer ermöglicht dies zudem eine Applikation als *Komposition* von Diensten zu formulieren. Weitere als Dienst realisierte Algorithmen, wie beispielsweise der in Kapitel 6 vorgestellte *DySSCo*-Algorithmus, eröffnen dem Nutzer zusätzliche Möglichkeiten, die Anforderungen an seine Anwendung zu formulieren. Die Komplexität des verteilten System wird somit vor dem Anwender vorborgen.

Die erarbeiteten Lösungen müssen dem Benutzer der Sensornetztechnologie jedoch zugänglich und nutzbar gemacht werden. So werden weiter Infrastrukturkomponenten benötigt, die es ermöglichen einzelne Dienste auszuwählen und in das Sensornetz migrieren zu lassen, damit Anwendungen komponiert werden können. Der Anwender hat damit die Möglichkeit, seine Anforderungen direkt an das Sensornetz zu stellen und Dienste – und somit Funktionalität – hinzuzufügen, zu ersetzen und wieder zu entfernen. Die Infrastruktur muss dabei so realisiert sein, dass es dem Benutzer zudem möglich ist, weitere Werkzeuge und Abstraktionen zur Applikationsentwicklung zu nutzen, die auf dem Paradigma der Service-Orientierung aufbauen. Dies kann beispielsweise die Integration der Prozessbeschreibungssprache wie BPEL (*Business Process Execution Language)* [77]

sein, die bereits im Bereich der verteilten Geschäftsanwendungen zur Modellierung bzw. Programmierung verwendet wird, oder auch die am Institut für Telematik der Universität zu Lübeck speziell für Sensornetze entwickelten Sprache *GWELS* (*Graphical Workflow Execution Language for Sensor Networks*) [27].

8.2 Architektur

Aus den genannten Anforderungen lassen sich drei Komponenten ableiten, die in einer Infrastruktur zur Anwendungsentwicklung interagieren müssen. Die Komponenten dargestellt in Abbildung 8.2 sind die Sensornetzanwendung, eine Servicekompositionskomponente und ein oder mehrere Dienstverzeichnisse.

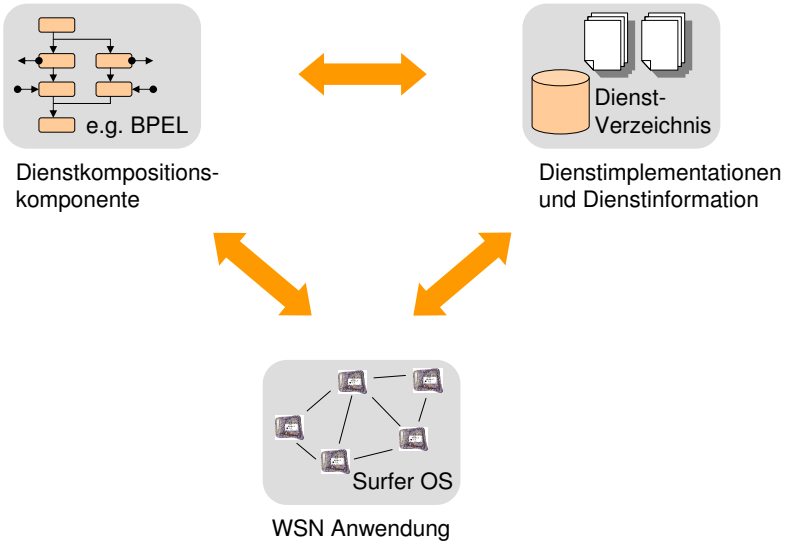


Abbildung 8.2: Komponenten der Infrastruktur zur service-orientierten Erstellung von Sensornetzapplikationen

Die Sensornetzapplikation besteht dabei initial lediglich aus dem Betriebssystem *Surfer OS*, das auf allen Sensorknoten vorhanden ist. Es bietet dabei die in Kapitel 4 beschriebenen Funktionalitäten an, nämlich den Zugriff auf die auf dem Knoten präsenten Hardwareressourcen und die Möglichkeit, weitere Dienste einzubinden.

Das Dienstverzeichnis enthält Algorithmen und Protokolle in Form von Diensten für *Surfer OS*. Diese Dienste werden von Entwicklern mit Expertise in verteilten Systemen und Sensornetzen bereitgestellt. Der Benutzer des Sensornetzes kann aus dem Dienstverzeichnis Dienste auswählen und in sein Sensornetz migrieren.

Die Dienstkompositionskomponente ermöglicht dem Benutzer, die Komposition der Dienste zu koordinieren. Hier wird festgelegt, wie und nach welchen Kriterien die Dienste auf

die Knoten im Sensornetz gelangen. Dabei können ebenfall komplexere Werkzeuge wie die zuvor erwähnte Prozessbeschreibungssprache BPEL zum Einsatz kommen.

Im Folgenden wird nun genauer auf das Dienstverzeichnis und die Dienstkompositionskomponente eingegangen.

8.2.1 Dienstverzeichnis

Kernkomponente eines Dienstverzeichnisses ist eine Datenbank, welche die Dienste in Maschinencode zusammen mit den Relokationsinformationen, die in Kapitel 5 beschrieben wurden, enthält. Zusätzlich werden Informationen zu den Diensten, wie beispielsweise eine Dienstbeschreibung und Informationen über den Autor des Dienstes in der Datenbank vorgehalten. Informationen wie der global eindeutige Servicename und die eindeutige Serviceidentifikationsnummer sowie die Versionsnummer, die von Seiten des *Surfer OS* benötigt werden (Kapitel 4.4) und im Maschinencode enthalten sind, werden dabei aus dem Code extrahiert. Wie in Kapitel 4.4 beschrieben hängt von diesen Informationen ab, wie *Surfer OS* den empfangenen Dienst auf den Sensorknoten handhabt.

Das Dienstverzeichnis bietet daher zwei weitere Komponenten: eine Datenbankzugangskomponente und eine Dienstmodifikationskomponente. Die Datenbankzugangskomponente ermöglicht es, Dienste in das Dienstverzeichnis abzuspeichern, aus dem Verzeichnis zu löschen und aus dem Verzeichnis zu laden, um sie beispielsweise in das Sensornetz zu übertragen. Die Dienstmodifikationskomponente ermöglicht dem Autor eines Dienstes, nachträglich Anpassungen der Dienstinformationen vorzunehmen. Somit kann der Maschinencode modifiziert werden, ohne diesen neu zu übersetzen und in das Dienstverzeichnis zu laden, was die Handhabung der Dienste vereinfacht.

8.2.2 Dienstkompositionskomponente

Die Dienstkompositionskomponente ist die Komponente, die der Anwendungsentwickler verwendet, um Dienste aus dem Dienstverzeichnis auszuwählen und in das Sensornetz zu migrieren. Sie stellt somit das Bindeglied zwischen Applikationsentwickler, Dienstverzeichnis und Sensornetz dar. Dabei unterteilt sich die Dienstkompositionskomponente in zwei weitere Komponenten: Migrationskomponente und Migrationskontrolle.

Die Migrationskomponente baut zum einen eine Verbindung zum Dienstverzeichnis bzw. zu der Datenbankzugangskomponente auf. Dadurch können Dienste aus dem Dienstverzeichnis geladen werden. Zum anderen wird die Verbindung zum Sensornetz hergestellt. Hierbei muss einerseits die hardwareseitige Verbindung zwischen Rechner und Sensornetz durch die Migrationskomponente angesprochen werden. Mittels einer speziellen RS232 (bzw. EIA-232) Steckverbindung lassen sich die *Pacemate*-Sensorknoten mit einem seriellen Port (COM-Port) des Rechners verbinden. Außerdem muss die Migrationskomponente die Relokationsinformationen und den Maschinencode eines Dienstes für das Kommunikationsprotokoll zwischen Rechnern und Sensorknoten aufbereiten. So

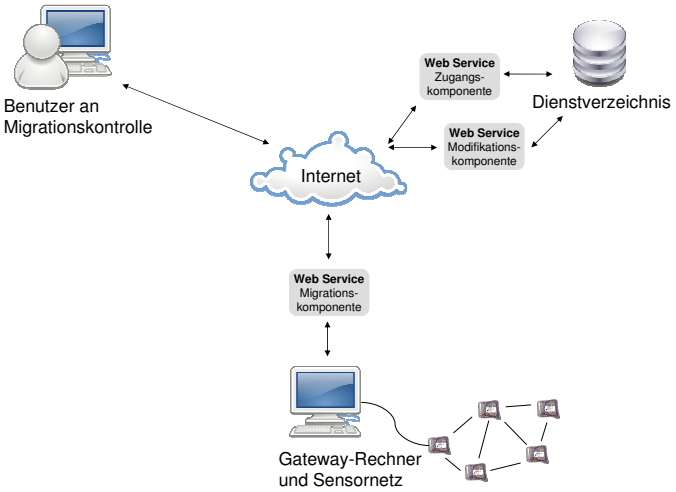


Abbildung 8.3: Implementierung der Architektur zum entwicklerorientierten Anwendungsdesign mittels Web Services

können beispielsweise nur Pakete mit eine maximalen Größe von 100 Byte gegebenenfalls über Funk weitergeleitet werden. Die Migrationskomponente zerlegt daher die zu übertragenden Daten bereits in Pakete dieser Größe und versieht diese mit notwendigen Paketinformationen wie einer Identifikationsnummer und einer Sequenznummer, so dass diese beim Empfänger wieder zusammengesetzt werden können.

Die Migrationskontrolle erlaubt dem Benutzer, die Migration zu steuern. Durch die Trennung von Migrationskomponente und Migrationskontrolle ist es möglich, hier verschiedene Werkzeuge einzusetzen, welche die zu migrierenden Dienste auswählen. Solche Werkzeuge können beispielsweise dem Benutzer eine abstrakte Sicht auf das Sensornetz anbieten, was diesem die Erstellung von Anwendungen vereinfacht. So können durch die Verwendung von Prozessbeschreibungssprachen Anwendungen entworfen werden. Bei der Ausführung werden auf dieser Basis die entsprechenden Dienste in das Sensornetz migriert.

8.3 Implementierung

In der Architektur sind die einzelnen Komponenten der Infrastruktur wie das Dienstverzeichnis und die Dienstkompositionskomponente sowie deren Unterkomponenten beschrieben worden. Da es sich bei dieser beschriebenen Infrastruktur zur Entwicklung von Sensornetzapplikationen selbst um eine verteilte Anwendung handelt, ist diese ebenfalls mittels des service-orientierten Paradigmas unter Verwendung der Web Service-Technologie realisiert worden. So sind im Dienstverzeichnis die Datenbankzugangskomponente und die Dienstmodifikationskomponente und in der Dienstkompositionskomponente die Migrationskomponente jeweils ein Web Service. Die resultierende Umsetzung

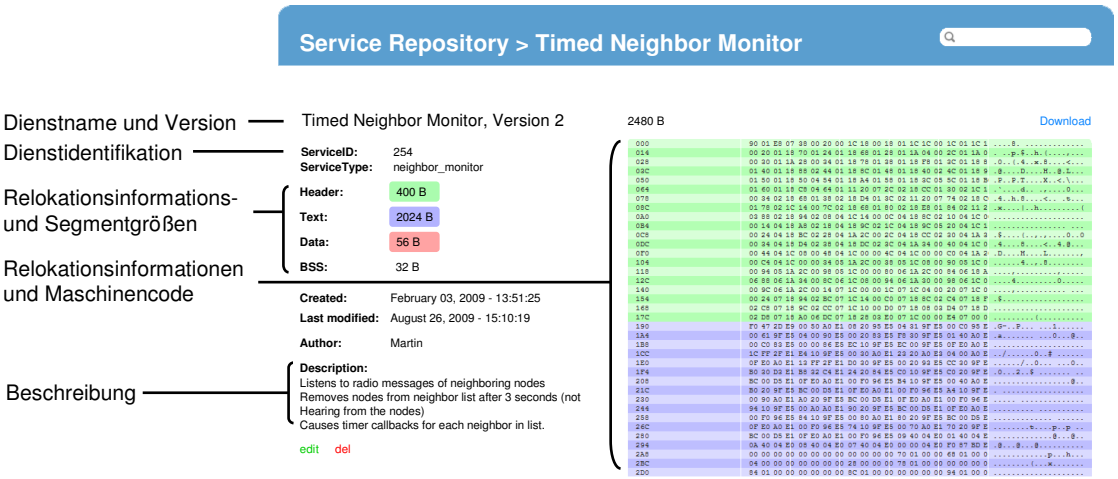


Abbildung 8.4: Internetseite zum Verwalten der Dienste im *Service Repository* durch den Dienstentwickler

und die Interaktion der Komponenten ist in Abbildung 8.3 dargestellt. Diese Realisierung bietet zusätzliche Flexibilität, auf welchen Rechnern sich das Dienstverzeichnis und die Verbindung zum Sensornetz befinden.

Ein Programmierer von Diensten für das Sensornetz kann über eine Internetseite dargestellt in Abbildung 8.4 Dienste in das Dienstverzeichnis einfügen, daraus löschen oder darin modifizieren. Die Internetseite stellt dabei die Inhalte des Dienstverzeichnisses dar und realisiert ein Formular, mit dem Aktionen wie Einfügen oder Löschen angestoßen werden können. Bei der Aufbereitung der Information werden die in Kapitel 5 eingeführten Relokationsinformationen sowie die Segmente TEXT und DATA farblich hervorgehoben und deren Größen angezeigt. Zur Ausführung der Aktionen nutzt die Seite dann die Web Services des Dienstverzeichnisses.

Für den Nutzer der Sensornetztechnologie wurde eine grafische Benutzerschnittstelle zur Migrationskontrolle realisiert, die in Abbildung 8.5 dargestellt wird. Diese Benutzerschnittstelle ist eine lokale Applikation, die in einer .Net-Implementation und einer Java-Implementation vorliegt und lokal auf dem Rechner ausgeführt wird.

Mittels dieser Benutzerschnittstelle kann der Nutzer einzelne Dienste aus einem Dienstverzeichnis auswählen und auf einzelne oder mehrere Knoten im Sensornetz migrieren. Die Applikation verbindet sich dazu mit dem in der URL-Zeile angegebenen Datenbankzugangs-Web Service eines Dienstverzeichnisses und mit dem Migration-Web Service, der auf dem Rechner mit Verbindung zum Sensornetz ausgeführt wird. Der Benutzer kann durch die Angabe einer Knotenadresse auswählen, ob er den Dienst auf einen einzelnen Knoten oder auf alle erreichbaren Knoten überträgt. Mittels eines Auswahlkastens (engl.: *checkbox*) in der Benutzerschnittstelle kann der Nutzer durch die Option *Enable Multihop* ein Weiterleitungsprotokoll auf den Knoten aktivieren, welches den Dienst durch

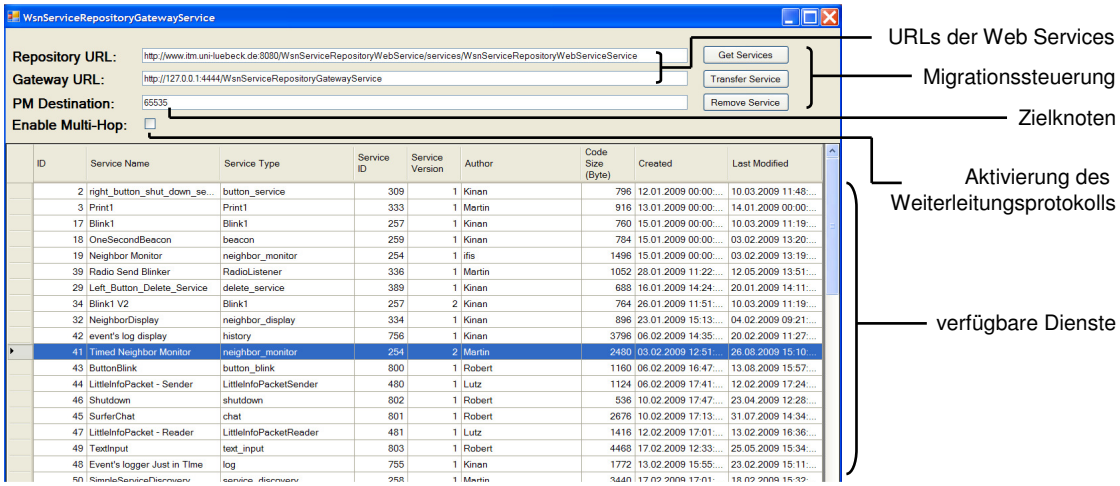


Abbildung 8.5: Grafische Benutzerschnittstelle zur Migrationskontrolle durch den Anwendungsentwickler

das gesamte Sensornetz flutet, um somit den angegebenen Knoten oder alle Knoten zu erreichen.

8.4 Service-orientierte Sensornetzapplikationen

Im Rahmen dieser Arbeit wurden etliche Dienste erstellt und verschiedene Anwendungen komponiert. Um die Interaktionen zwischen den Komponenten der service-orientierten Infrastruktur – nämlich dem Sensornetz bestehend aus *Pacemate*-Sensorknoten mit *Surfer OS*-Betriebssystem, dem Dienstverzeichnis und der Dienstkompositionskomponente – zu verdeutlichen, werden im Folgenden drei mittels des service-orientierten Programmierparadigmas erstellte Sensornetzanwendungen vorgestellt. Die Applikationen und der Erstellungsprozess sind dabei insbesondere unter dem Aspekt zu betrachten, wie ohne Fachkenntnisse über Sensornetze sowie ohne Programmierkenntnisse eine Anwendung mittels der in dieser Arbeit vorgestellten Ergebnisse aus einzelnen Komponenten zusammengesetzt werden kann.

8.4.1 Fallbeispiel 1: Nachbarschaftsmessung

Viele Algorithmen basieren auf Nachbarschaftsinformationen. Zum einen werden Entscheidungen basierend auf der Größe der direkten Nachbarschaft getroffen, zum anderen ist es zudem für manche Algorithmen und Anwendungen wichtig zu wissen, welche Knoten sich in der direkten Nachbarschaft befinden. In dieser Beispielanwendung werden mittels der Kombination weniger Dienste die Informationen über die direkten Nachbarn auf den Knoten bereitgestellt und zur Demonstration auf dem Bildschirm der *Pacemate*-Sensorknoten ausgegeben.

Zunächst wurden die folgenden Dienste implementiert und im Dienstverzeichnis abgelegt: ein Nachbarschaftsmonitordienst, ein Beacon-Dienst und ein Nachbarschaftsanzeigedienst.

Der Nachbarschaftsmonitor prüft alle empfangenen Nachrichten. Die Absender der Nachrichten werden in einer Nachbarschaftsliste abgelegt. Um alle Nachrichten zu empfangen, registriert sich der Nachbarschaftsmonitor beim Aufruf seiner `ServiceStart()`-Funktion direkt bei der Funkschnittstelle im *Surfer OS* (nicht bei dem Funkprotokollstapel). Der Nachbarschaftsmonitor veröffentlicht zwei Funktionen bei dem in Kapitel 4 eingeführten *ServiceManager*, über die man zum einen die Größe der Nachbarschaft und zum anderen eine Liste mit Knotenidentifikationsnummern erhält.

Der Beacon-Dienst sendet im festgelegten Zeitintervall (1 Sekunde) eine Nachricht. Zum Versenden einer Nachricht nutzt der Beacon-Dienst direkt die Sendefunktionalität des Funks in *Surfer OS* und nicht den Protokollstapel. Für das Einhalten des Sendeintervalls nutzt der Beacon-Dienst einen Dienst des im *Surfer OS* vorhandenen *Task Managements*, das erlaubt, eine Rückruffunktion zu registrieren, die nach Ablauf eines gegebenen Zeitintervalls vom *Task Management* aufgerufen wird.

Der Nachbarschaftsanzeigedienst zeigt die aktuelle Nachbarschaftsgröße des Knotens auf der Anzeige des *Pacemate*-Sensorknotens an. Der Dienst nutzt dazu den Nachbarschaftsmonitordienst.

Um die Anwendung zu komponieren und zum Einsatz zu bringen, werden zunächst verschiedene *Pacemate*-Sensorknoten mit *Surfer OS* zum Einsatz gebracht. Ein beliebiger Knoten wird über die RS232-Verbindung mit einem Rechner verbunden. Über die grafische Benutzerschnittstelle zur Migrationskontrolle kann der Benutzer alle Dienste aus dem Dienstverzeichniss anwählen. Zunächst wird der Nachbarmonitordienst ausgewählt. Als Zielknoten wird die *Broadcast*-Adresse der *Pacemate*-Sensorknoten 65535 ($0xFFFF_{16}$) eingesetzt. Je nach Verteilung der Knoten muss der Auswahlkasten zur Multi-hop-Weiterleitung aktiviert werden, so dass alle Knoten im ausgebrachten Netz die Dienste erhalten. Über die Schaltfläche *Transfer Service* wird der Dienst in das Netz migriert. Die beiden anderen Dienste werden auf die gleiche Weise auf alle Knoten migriert. Die Knoten zeigen nun die Anzahl ihrer jeweiligen direkten Nachbarn auf ihrer Anzeige an.

8.4.2 Fallbeispiel 2: Multi-hop-Demonstration

In Kapitel 4 wurde beschrieben, dass lediglich die Hardwareschnittstellen im *Surfer OS*-Betriebssystem als Dienste angeboten werden und dass es möglich ist, Protokolle, die beispielsweise den Funkkommunikationsstapel (engl.: *radio stack*) bilden, auszutauschen. Um diese Flexibilität zu demonstrieren, wird eine Anwendung komponiert, bei der der *radio stack* ausgetauscht wird. Zu Demonstrationszwecken können in dieser Applikation per Tastendruck von jedem *Pacemate*-Sensorknoten zwei verschiedene Nachrichten abgesetzt werden. Bei der einen Nachricht schalten alle empfangenden Knoten die LED an, bei der anderen schalten alle empfangenden Sensorknoten die LED aus. Damit kann

gezeigt werden, ob die Nachricht mittels eines Wegewahlverfahrens nach Austausch des *radio stacks* durch das Netz weitergeleitet wird.

Diese Anwendung setzt sich aus drei Diensten zusammen: ein Wegewahldienst, ein Radio-Dienst und eine LED-Control-Dienst.

Der hierfür verwandte Wegewahlservice realisiert das Fluten eines Paketes durch das Netz. Wird ein Paket mittels dieses Dienstes versandt, wird ein Paketkopf mit einer Paket ID, einer Absenderadresse und einer Zieladresse hinzugefügt. Wird ein Paket über diesen Dienst empfangen, so wird anhand der Paket ID und Absenderadresse überprüft, ob dieses Paket von diesem Knoten bereits empfangen wurde. Ist dies nicht der Fall, werden diese beiden Werte gespeichert und das Paket wird verarbeitet. Bei der Verarbeitung wird anhand der Zieladresse entschieden, ob das Paket für diesen Knoten bestimmt ist. In diesem Fall wird das Paket an die beim *radio stack* registrierten Dienste ausgeliefert. Ist der Knoten nicht die Zieladresse, so wird das Paket weitergesandt.

Der Radio-Dienst stellt eine neue Implementierung des *radio stacks* dar und ersetzt die auf dem Knoten vorhandene Implementierung. Der auf dem Knoten vorhandene *radio stack* liefert die über die Funkschnittstelle empfangenen Pakete direkt an registrierte Dienste. Der hier verwendete *radio stack* gibt die über die Funkschnittstelle empfangenen Dienste zunächst an den Wegewahlservice weiter, bei dem *radio stack* mit einer speziellen Funktion auch als empfangender Dienst registriert ist. Wird diese Funktion seitens des Wegwahldienstes aufgerufen, liefert der Radio-Dienst dieses Paket an die bei ihm registrierten Applikationen. Zum Versenden von Paketen wird ebenfalls zunächst der Wegewahldienst aufgerufen, der den entsprechenden Paketkopf für das Paket erstellt.

Der LED-Control-Dienst versendet Nachrichten an die *Broadcast*-Adresse. Diese Nachrichten ändern den Status der LED auf den empfangenden Knoten. Wird auf die linke Taste gedrückt, so wird eine Nachricht erzeugt, welche die LED auf den empfangenden Geräten einschaltet. Mittels der rechten Taste wird die LED auf den Empfängern ausgeschaltet. Der LED-Control-Dienst registriert sich beim *radio stack*. Im Vergleich: der Nachbarschaftmonitordienst aus der Nachbarschaftsmessungsanwendung registrierte sich direkt bei der Funkschnittstelle.

Zunächst wird der LED-Control-Dienst auf alle Knoten migriert. Beim Ausführen der Anwendung kann man sehen, dass mittels des LED-Control-Dienstes zunächst nur die Knoten in der direkten Nachbarschaft auf die Nachrichten durch Ein- und Ausschalten der LED reagieren. Erst nach dem Migrieren des Wegewahlservices und des Radio-Dienstes auf alle ausgebrachten Knoten werden die Nachrichten des LED-Control-Services über alle Knoten (multi-hop) weitergeleitet. Dadurch reagieren nun alle Knoten im Netz auf den Tastendruck eines Knotens.

8.4.3 Fallbeispiel 3: Wandernder zustandsbehafteter Dienst

In Kapitel 5 wurde beschrieben, dass es möglich ist, einen Dienst zustandsbehaftet mit den aktuellen Werten der Variablen des Dienstes migrieren zu können. Um dies zu demonstrieren, wird eine Anwendung komponiert, in der es dem Benutzer möglich ist, den

Status eines Dienstes, hier dargestellt durch die Blinkfrequenz der LED, zu verändern. Der Dienst migriert auf einen Nachbarknoten und behält dabei seinen Status.

Diese Anwendung setzt sich aus drei Diensten zusammen: Nachbarschaftsmonitordienst, Beacon-Dienst und Wanderdienst.

Der Nachbarschaftmonitordienst und der Beacon-Dienst wurden bereits in der ersten Beispielanwendung in Kapitel 8.4.1 vorgestellt. In dieser Applikation werden die beiden Dienste von dem Wanderdienst dazu genutzt, den nächsten Zielknoten, auf den der Service migriert, auszuwählen.

Der Wanderdienst erzeugt zunächst ein Blinken der LED. Mittels rechter und linker Taste des *Pacemates* kann der Benutzer die Frequenz erhöhen oder verringern. Der Wanderdienst zeigt dabei die bereits *besuchten* Knoten auf der Anzeige an und sucht sich bereits den nächsten Zielknoten. Durch Betätigen der mittleren Taste wandert der Wanderdienst auf den nächsten Knoten und löscht sich auf dem bisherigen. Dabei behält er die eingestellte Blinkfrequenz der LED bei.

Der Nachbarschaftsmonitordienst sowie der Beacon-Dienst werden zunächst mittels der GUI auf alle Knoten migriert. Anschließend wird der Wanderdienst auf einen einzelnen Knoten migriert. Der Benutzer interagiert nun mit diesem *Pacemate*-Knoten, auf dem sich der Wanderdienst befindet. Drückt der Benutzer die mittlere Taste, so wandert der Dienst auf den nächsten Knoten. Hier ist wichtig, sich zu verdeutlichen, dass bei diesem Vorgang der Programmcode des Dienstes samt der aktuelle Werte der Variablen auf einen anderen Knoten wandert und dort weiter ausgeführt wird.

8.5 Ergebnis

In diesem Kapitel wurde eine Infrastruktur vorgestellt, welche das durch das *Surfer OS*-Betriebssystem umgesetzte Paradigma der Service-Orientierung im Hinblick auf die Anwendungserstellung für Sensornetze unterstützt. Die Infrastruktur gibt dem Anwender ohne Sensornetzexpertise die Möglichkeit, seine Anwendung in abstrakter Form zu formulieren, und setzt diese im Sensornetz um. Durch die modulare Aufteilung in Dienstverzeichnisse mit Implementationen verschiedener Services für *Surfer OS* und eine Servicekompositionskomponente sowie der Standardtechnologie der Web Services ist es dem Benutzer zudem möglich, verschiedene Werkzeuge zur abstrakten Anwendungsbeschreibung in die Infrastruktur einzubinden.

Gleichzeitig wird den Sensornetzexperten die Möglichkeit gegeben, Dienste zu implementieren und diese in einem Dienstverzeichnis zur Verfügung zu stellen. Der Dienstentwickler kann mittels der zur Verfügung gestellten Webschnittstelle Dienste verändern, ohne sie neu übersetzen zu müssen.

Die für den Benutzer der Sensornetztechnologie gegebene Abstraktion der Service-Orientierung wird in der hier präsentierten Realisierung über eine graphische Benutzerschnittstelle angeboten. Der Anwender kann beliebige Dienste auszuwählen, auf die Knoten im

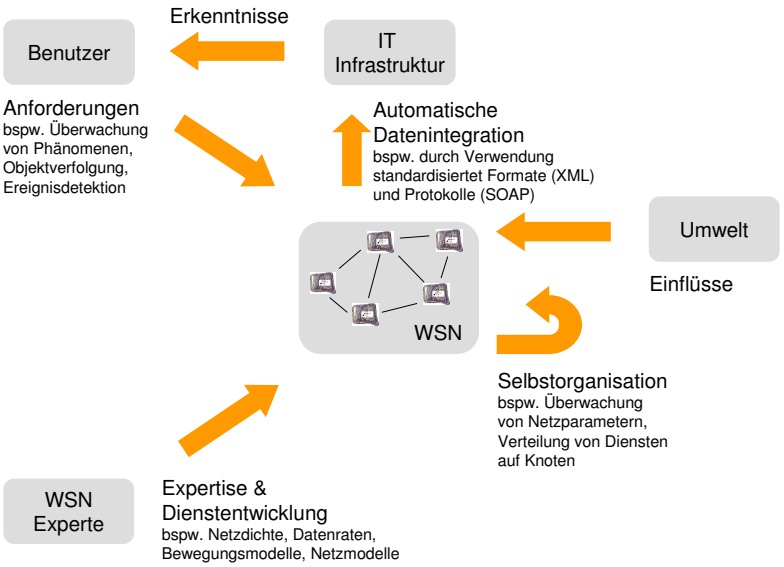


Abbildung 8.6: Applikationsentwicklungszyklus mittels SOA und Entwicklungsabstraktionen

Netz migrieren und so individuelle Anwendungen komponieren. Zudem ist es dem Anwender möglich, die Dienste über dieselbe Schnittstelle wieder zu entfernen und so seine Anwendung anzupassen oder zu korrigieren.

Die Einfachheit des so umgesetzten Entwicklungsprozess zu evaluieren und zu bewerten, wird als schwierig betrachtet. Fakt ist jedoch, dass ein aus den verteilten Systemen bekanntes Paradigma als Abstraktion für den Benutzer umgesetzt wurde. Der Nutzer kann eigenständig Funktionalität hinzufügen oder entfernen, ohne Programmcode erstellen zu müssen. Der Nutzer muss sich keine Gedanken um Implementierungsdetails, Hardwaredetails, Zeitsteuerung (engl.: *timing*), Wettlaufsituationen o.ä. machen. Durch die umgesetzte Infrastruktur bleiben diese Probleme aus dem Bereich der verteilten Systeme für den Benutzer transparent.

Mittels des Paradigmas der Service-Orientierung und der vorgestellten Infrastruktur verändert sich auch der beobachtete Ablauf bei der Erstellung von Sensornetzanwendungen. Im Gegensatz zu dem bisher angenommenen Ablauf aus Abbildung 8.1 stellt sich der Prozess der Anwendungsentwicklung auf Grund des realisierten Konzeptes jetzt wie in Abbildung 8.6 dar. Der Benutzer muss nun seine Anforderungen nicht erst dem Sensornetzexperten schildern, sondern kann diese mittels der gegebenen Abstraktion direkt an das Sensornetz stellen. Mit Diensten, die die entsprechenden Schnittstellen realisieren, kann das Sensornetz auch leicht in eine bestehende Infrastruktur eingebettet werden. Durch die mittels des service-orientierten Paradigmas realisierte lose Kopplung der Anwendung können diese Schnittstellen leicht ersetzt und ausgetauscht werden. Der Sensornetzexperte stellte die verschiedenen Dienste, die der Nutzer verwenden möchte, über ein Dienstverzeichnis dem Benutzer zur Verfügung. Unter Verwendung von ver-

teilten Algorithmen zur Selbstorganisation wie beispielsweise den *DySSCo*-Algorithmus kann das Sensornetz auch eigenständig Anpassungen vornehmen und beispielsweise auf Einflüsse aus der Umwelt reagieren.

9 Zusammenfassung und Ausblick

Computer sind in der heutigen Zeit allgegenwärtig. Im Zuge der Miniaturisierung und Vernetzung verschmelzen Kleinstcomputer mit ihrer Umwelt und haben die Möglichkeit, miteinander zu interagieren, um kooperativ komplexe Aufgaben zu lösen. Aus dieser immer noch fortschreitenden Entwicklung im *ubiquitous* und *pervasive Computing* hat sich – neben anderen – die Technologie der Sensornetze etabliert und Raum für neue Anwendungen geschaffen. Wie jedoch auch zu Beginn des Computerzeitalters ist diese neue Technologie nur Nutzern zugänglich, die über Fachkenntnisse verfügen. Die notwendigen Kenntnisse müssen hierbei sowohl im Bereich der verteilten Systeme wie auch der eingebetteten Systemen liegen. Die Eigenschaft der Verteiltheit sowie der Nebenläufigkeit gepaart mit den Eigenheiten der Funkübertragung und den speziellen Rahmenbedingungen der stark begrenzten Ressourcen auf den Sensorknoten erfordern ein hohes Maß an Expertise und Erfahrung, um Sensornetzanwendungen zu erstellen. Auch dann erfordert der Entwurf einer Sensornetzanwendung zunächst oft aufwendige Messungen vor Ort sowie komplizierte Modellbildungen. Nachträgliche Anpassungen sowie Erweiterungen von Anwendungen sind ebenfalls aufwendig und können ebenfalls nur von den Experten vorgenommen werden. Hinzukommt, dass für viele der in der Forschung präsentierten Lösungen zu einzelnen Teilproblemen wie beispielsweise der Wegewahl der Beweis der Praxistauglichkeit noch aussteht. Aus finanziellen, organisatorischen sowie zeitlichen Gründen werden Algorithmen und Protokolle oft lediglich theoretisch oder simulativ untersucht.

Der Beitrag dieser Arbeit ist es, die Sensornetztechnologie den potentiellen Anwendergruppen zugänglich zu machen, indem das service-orientierte Paradigma in den Bereich der Sensornetze übertragen und an dessen spezielle Rahmenbedingungen angepasst wurde. Hierdurch sind zum einen Möglichkeiten geschaffen worden, die Komplexität der Sensornetze als hochgradig verteiltes System vor dem Anwender zu verbergen. Zum anderen wurde die Flexibilität umgesetzt, die nötig ist, um jederzeit Modifikationen an laufenden Sensornetzanwendungen vorzunehmen und dadurch zeitnah auf sich ändernde Bedingungen und Anforderungen reagieren zu können. Der Autor hat eine Infrastruktur entwickelt bestehend aus einem Betriebssystem, einer Technik zur Dienstmigration und Werkzeugen zur Applikationsentwicklung. Zusätzlich wurden robuste Algorithmen entworfen, die speziell im Hinblick auf die Verteiltheit und Dynamik eines Sensornetzes entwickelt und untersucht wurden und deren Praxistauglichkeit für den Einsatz in Sensornetzen experimentell verifiziert wurde.

Nach einer Motivation und Einführung in die Grundlagen zu Sensornetzen und Service-orientierten Architekturen in Kapitel 1 und Kapitel 2 wurde in Kapitel 3 die im Rahmen

der Arbeit entwickelte Sensorknotenplattform *Pacemate* vorgestellt. Durch die Einbettung in ein robustes Gehäuse eröffnet die *Pacemate*-Plattform experimentelle Möglichkeiten unter Nicht-Laborbedingungen. So ist es möglich, die Geräte an etliche Personen auszugeben, ohne dass diese besondere Rücksicht auf die pflegliche Handhabung der Technik der Geräte nehmen müssen. Dies ermöglicht beispielsweise die Evaluation von Algorithmen und Protokollen in mobilen Szenarien mit vielen Knoten. Die *Pacemate*-Plattform verfügt zudem über Tasten und eine grafische Anzeige. Dadurch ist es möglich, den genauen Status des Gerätes sowie die Werte von Variablen zu modifizieren und abzuprüfen. Somit lässt sich die Arbeitsweise von Algorithmen und Protokolle sowie die Interaktionen zwischen den einzelnen Sensorknoten in der Praxis verifizieren. Die *Pacemate*-Plattform stellt dadurch die Grundlage zur Entwicklung von komplexen praxistauglichen Anwendungen dar.

Auf Basis der Sensorknotenplattform *Pacemate* wurde in Kapitel 4 das neuartige Betriebssystem *Surfer OS* für Sensorknoten entwickelt. Basierend auf der Grundidee, jegliche Funktionalität als ersetzbare Dienste zu betrachten, wurde das service-orientierte Paradigma nicht nur für Sensornetzanwendungen oberhalb des *Surfer OS* sondern auch innerhalb des Betriebssystems selbst realisiert. So stellt das *Surfer OS* als minimales Betriebssystem lediglich die notwendigen Funktionalitäten wie Hardwareschnittstellen, Task Management, Speicher Management und Service Management zur Verfügung. Durch diesen Ansatz wird den Anforderungen der hohen Flexibilität, die anwendungs- und szenarioabhängig in allen Schichten des Systems benötigt wird, Genüge getan. So ist es dadurch möglich, während des Betriebes des Sensornetzes oberhalb der Hardwareschnittstellen Dienste hinzuzufügen oder basierend auf neuen Anforderungen und Topologieänderungen auszutauschen und so das Sensornetz anzupassen. Dieser Ansatz der losen Kopplung des Betriebssystems und der Anwendungskomponenten steht damit entgegen dem Ansatz des weitverbreiteten Sensorknotenbetriebssystems TinyOS, bei dem das Betriebssystem zusammen mit der Anwendung übersetzt und gebunden wird und eine starre eng gekoppelte Applikation erstellt wird.

Um Dienste in *Surfer OS* hinzuzufügen und ersetzen zu können, wurde in Kapitel 5 ein plattformunabhängiges Vorgehen vorgestellt, dass es ermöglicht, unabhängig übersetzte Programme in Form von Diensten zu migrieren und in eine laufende Applikation einzubinden. Dabei werden die vom Linker zur Verfügung gestellten Relokationsinformationen komprimiert aufbereitet und bei der Migration des Programmes mitgeführt. Basierend auf diesen Informationen sowie den zugewiesenen Speicheradressen werden beim Empfänger jegliche Adressierungen innerhalb des empfangenen Maschinencodes neu berechnet, so dass das Programm fehlerfrei ausgeführt werden kann. Durch die Vereinbarung eines jedem Programm zu Grunde liegenden Programmgerüstes wird ein Einstiegspunkt in das Programm festgelegt, über den dieses auf dem Empfänger ausgeführt wird. Das Verfahren wurde innerhalb des *Surfer OS*-Betriebssystems auf der *Pacemate*-Plattform exemplarisch realisiert und eingesetzt.

Die Kapitel 4 und Kapitel 5 realisieren mit dem Betriebssystem *Surfer OS* und der Migration von Diensten das service-orientierte Paradigma innerhalb eines Sensornetzes. Dies eröffnet zum einen dem Entwickler von Sensornetzanwendungen eine neue Abstraktion

im Sinne der Service-Orientierung. Zum anderen schafft dies auch weitere Möglichkeiten zur Abstraktion, die dem Entwickler neue Wege bieten, Anforderungen an Anwendungen zu formulieren. Insbesondere können auf dieser Basis Algorithmen zur Selbsorganisation entwickelt werden, welche die Migration von Diensten basierend auf Netzparametern koordinieren. Dies ist ein weiterer Schritt, die Komplexität des verteilten Sensornetzes zu verbergen und somit die Applikationserstellung zu vereinfachen.

Unter der Vorgabe, Komplexität zu verbergen und gleichzeitig den Anforderungen der Praxistauglichkeit und Robustheit zu genügen, wurde in Kapitel 6 mit dem *DySSCo*-Algorithmus ein Verfahren vorgestellt, welches es erlaubt, Anforderungen an eine serviceorientierte Applikationen durch prozentuale Abdeckungen von Diensten zu formulieren. Der *DySSCo*-Algorithmus platziert die Dienste im Netz so, dass die geforderte Abdeckung erfüllt wird, und sorgt dafür, dass diese auch nach Topologieänderungen erhalten bleibt. Dabei werden lediglich lokale Entscheidungen auf jedem Knoten getroffen. Es wurde gezeigt, dass durch das lokale Verhalten ein global emergentes Verhalten erreicht werden kann, so dass die geforderte Abdeckung im Netz erfüllt wird. Obwohl gezeigt wurde, dass der Algorithmus ohne Erweiterung in wenigen gerichteten zyklischen Graphen keinen stabilen Endzustand besitzt, bewies der *DySSCo*-Algorithmus seine Praxistauglichkeit in praktischen Einsätzen. Der Algorithmus ist auf 20 *Pacemate*-Sensorknoten als eigener Service auf dem *Surfer OS*-Betriebssystem realisiert worden. Es konnte demonstriert werden, dass der Algorithmus auch unter Nicht-Laborbedingungen mit konstantem Nachrichtenaufkommen die geforderten Anforderungen erfüllt und somit praktisch einsetzbar ist. Diese Art der Verifikation der Praxistauglichkeit im Rahmen eines Experimentes auf Sensorknoten wurde bei den verwandten Arbeiten nicht erbracht.

Unter den selben Anforderungen der Praxistauglichkeit und Robustheit wurde in Kapitel 7 das Wegewahlverfahren *GRAPE* entworfen. Auf Grund der gegebenen Rahmenbedingungen der schwankenden Kommunikationsverbindungen in Sensornetzen löst sich der *GRAPE*-Algorithmus von der Idee, Pfade zwischen Quelle und Senke aufzubauen, auf der viele Wegewahlverfahren basieren. In *GRAPE* werden die Pakete entlang eines Gradienten zur Senke weitergeleitet. Dabei entscheiden die empfangenden Knoten, ob sie das Paket weiterleiten. Insbesondere wird bei *GRAPE* die Eigenschaft des Funkes, dass alle Nachrichten innerhalb einer direkten Nachbarschaft empfangen werden, ausgenutzt. Auf diese Weise wird durch Interpretation von allen Nachrichten auf dem Medium sowohl permanent Informationen über die Netztopologie von den Knoten gesammelt als auch von jedem Knoten autark entschieden, ob noch die Notwendigkeit besteht, eine Nachricht weiterzuleiten. Der *GRAPE*-Algorithmus basiert dabei auf einfachen lokalen Regeln und wurde ebenfalls als Dienst für *Surfer OS* realisiert. Die simulativen Untersuchungen zeigen, dass der *GRAPE*-Algorithmus auch in hochgradig mobilen Netzen Auslieferungsraten von über 90 % liefert bei gleichzeitig im Vergleich zu AODV geringen Verzögerungszeiten. Die Praxistauglichkeit des Verfahrens für das aus der simulativen Untersuchung nachgestellte Szenario konnte nicht gezeigt werden. Hier sind weitere Untersuchungen notwendig.

Durch die Übertragung des service-orientierten Paradigmas ändert sich insbesondere das Vorgehen bei der Applikationserstellung. Wie die Realisierungen der in Kapitel 6

und Kapitel 7 vorgestellten verteilten Algorithmen *DySSCo* und *GRAPE* zeigen, wird für das service-orientierte Betriebssystem jegliche Funktionalität als Dienst umgesetzt. Insbesondere durch den *GRAPE*-Algorithmus wird deutlich, dass Dienste im *Surfer OS* sich gemäß des ISO-OSI Referenzmodells nicht nur auf der Applikationsebene befinden, sondern in jeder Schicht existieren können.

Um dem Anwendungsentwickler das service-orientierte Paradigma und die damit verbundene Flexibilität des *Surfer OS* zugänglich zu machen, wurde in Kapitel 8 eine Infrastruktur entwickelt, die ein entwicklerorientiertes Anwendungsdesign ermöglicht. Durch die Zerlegung in Sensornetz und Applikation, Dienstverzeichnis und Migrationskomponente wird dem Anwendungsentwickler eine Schnittstelle gegeben, mittels derer er aus vorhandenen Diensten auswählen und eine Anwendung im Sensornetz komponieren kann. Anhand von Beispielen konnte gezeigt werden, wie durch feingranulares Servicedesign verschiedenste Anwendungen über eine graphische Benutzerschnittstelle zur Migrationskontrolle zusammengesetzt werden können, ohne dabei Programmcode für die zu Grunde liegende Hardware der Sensorknoten erzeugen zu müssen. Dabei können verschiedene Dienste in immer neuen Zusammenhängen wiederverwendet werden. Die realisierte Architektur erlaubt dabei zudem das Einbinden von weiteren Werkzeugen zur service-orientierten Applikationserstellung, wie beispielsweise die Integration der Beschreibungssprache BPEL.

Die in den einzelnen Kapiteln vorgestellten Verfahren und Infrastrukturen lassen sich unabhängig weiterentwickeln. So wird in aktuellen Arbeiten am Institut für Telematik der Universität zu Lübeck das *Surfer OS* auf die Sensorknotenplattform TelosB übertragen. Weiterhin ist es möglich, *Surfer OS* um eine Typüberprüfung der Funktionssignaturen mittels des Ansatzes komprimierter Strings zu erweitern. Das *GRAPE*-Wegewahlverfahren kann in zukünftigen Arbeiten um weitere Metriken erweitert werden, welche die Auslastung einzelner Knoten berücksichtigen. So kann eine Fairness beim Weiterleiten gefunden werden, welche die Lebensdauer des Netzes erhöht. Aus den beiden vorgestellten Algorithmen *DySSCo* und *GRAPE* lassen sich gemeinsame Strategien identifizieren. Hier sind beispielsweise die Ausnutzung des Funkmediums durch aktives Abhören des Funkverkehrs und die lokale Entscheidungsfindung, die zu einem global emergenten Verhalten führt, zu nennen. Daher sollte untersucht werden, inwieweit sich aus diesen Algorithmen allgemeine Entwurfsstrategien für verteilte Algorithmen in Funknetzen ableiten lassen.

Als Gesamtheit stößt die Arbeit in den Bereich der Service-Orientierung in Sensornetzen vor und eröffnet hier neue weiterführende Möglichkeiten. Hier ergeben sich Fragestellungen zur Nutzung entfernter Dienste, zur Selbstorganisation und zur verteilten Prozessmodellierung.

In zukünftigen Arbeiten kann der Aspekt behandelt werden, wie Dienste so adressiert werden können, dass diese nicht nur lokal auf einem Knoten genutzt werden. So muss es möglich sein, dass ein Dienst einen zweiten Dienst nutzt, der sich auf einem anderen Knoten oder sogar außerhalb des Sensornetzes befindet. Dies erfordert zudem Konzepte zum Auffinden von Diensten in Sensornetzen. Hierbei ist interessant, inwieweit Techno-

logien wie beispielsweise XML, SOAP und Web Services zu diesem Zweck in Sensornetze übertragen werden können.

Eine weitere Perspektive bietet die Realisierung von selbstorganisierenden Strategien, welche die Transparenz für den Benutzer weiter erhöhen. Wie bereits durch den *DySSCo*-Algorithmus verdeutlicht wird, ergibt sich aus dem service-orientierten Ansatz die Möglichkeit, selbstorganisierende Strategien in ein System einzubetten. Diese Strategien platzieren und verteilen Dienste eigenständig im Sensornetz basierend auf Netzparametern, die für den Benutzer nicht sichtbar sind. Dabei ist ebenfalls noch offen, wie die Abhängigkeiten eines Dienstes zu Daten und anderen Diensten behandelt werden. Diese müssen zum einen in einer Dienstbeschreibung vorliegen, zum anderen bei der Migration eines Dienstes berücksichtigt werden. So müssen gegebenenfalls andere Dienste und Daten ebenfalls migrieren, sodass die Funktionalität auch nach der Migration weiterhin gegeben ist. Konzepte zur Replikation und zu Transaktionen, die innerhalb der Sensornetze realisiert werden, können hier die Konsistenz des Systems gewährleisten.

Ebenfalls von Interesse sind weitere Abstraktionen zur Anwendungsbeschreibung. In dieser Arbeit wurde bereits auf die Möglichkeit der Integration der Prozessbeschreibungssprache BPEL hingewiesen. Jedoch wird ein BPEL-Graph zentral außerhalb des Netzes ausgeführt. Im Hinblick auf eine mögliche Partitionierung des Netzes erscheint dies jedoch nicht als sinnvoll. So kann in weiteren Arbeiten untersucht werden, wie eine verteilte Prozessbeschreibung in Sensornetzen realisiert werden kann.

Die vorgelegte Arbeit stellt einen ersten Schritt zur transparenten Integration der Sensornetztechnologie in bestehende IT-Infrastrukturen dar. So wird durch das Paradigma der Service-Orientierung bereits von dem Sensornetz abstrahiert. Durch die Verwendung der vorgestellten Migrationskomponente ist das Sensornetz aus Sicht des Benutzer bereits dahingehend transparent, dass lediglich einzelne abstrakte Dienste in das Netz migriert oder aus dem Netz entfernt werden. Im nächsten Schritt nutzen die Dienste auf den Knoten nicht mehr nur andere Dienste auf dem gleichen Knoten, sondern Dienste, die sich auf anderen Knoten befinden. Durch die Integration und Verwendung von Standardtechnologien wie beispielsweise XML und Web Services lösen sich die Grenzen zwischen Internet und Sensornetz gänzlich auf. So integriert der Anwender einen Dienst, dessen Dienstanbieter das Sensornetz ist, genauso in seine lokale Applikation wie jeden anderen Dienst, der sich auf einem beliebigen Rechner im Internet befindet. Umgekehrt benutzt das Sensornetz Dienste, die sich außerhalb des Sensornetzes im Internet befinden. Das Sensornetz wird letztendlich mit dem Internet verschmelzen.

Abkürzungsverzeichnis

AODV	Ad-hoc On-Demand Distace-Vector Routing
API	Application Programming Interface
ARM	Acorn RISC Machine
ASK	Amplitude Shift Keying
AVR	Advanced Virtual Risc
BPEL	Business Process Execution Language
BSS	Block Started by Symbol
CELF	Compact ELF
CERN	Conseil Européen pour la Recherche Nucléaire
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DARPA	Defense Advanced Research Projects Agency
DSDV	Destination-Sequenced Distance-Vector Routing
DySSCo	Dynamic Self-organizing Service Coverage
EEPROM	Electrically Erasable Programmable Read-Only Memory
EIA	Electronic Industries Alliance
ELF	Executable and Linking Format
ESB	Embedded Sensor Board
FSF	Free Software Foundation
FSK	Frequency Shift Keying
FTP	File Transfer Protocol
GCC	GNU Compiler Collection
GNU	GNU is not Unix
GPS	Global Positioning System
GPSR	Greedy Perimeter Stateless Routing
GRAB	GRAdient Broadcast
GRAPE	Gradient based Routing for all Purposes
GWELS	Graphical Workflow Execution Language for Sensor Networks
HAL	Hardware Abstraction Layer
HTTP	Hypertext Transfer Protocol
I^2C	Inter-Integrated Circuit
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers, Inc.
IIOP	Internet Inter-ORB Protocol

ISM Band	Industrial, Scientific and Medical Band
ISO	International Organization for Standardization
JMS	Java Message Service
LEACH-C	Low Energy Adaptive Clustering Hierarchy Centralized
LED	Light Emitting Diode
MANTIS	Multimodal Networks of In-situ Sensors
MMU	Memory Management Unit
OASIS	Organization for the Advancement of Structured Information Standards
OLSR	Optimized Link State Routing
ORB	Object Request Broker
OS	Operating System
OSI	Open Systems Interconnection
PAWS	Petroleum Application of Wireless Systems
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RMI	Remote Method Invocation
ROM	Read-Only Memory
RPC	Remote Procedure Call
RREP	Route Reply
RREQ	Route Request
RS	Recommended Standard
RSSI	Received Signal Strength Indication
SMTP	Simple Mail Transfer Protocol
SOA	Service-orientierte Architektur
SPI	Serial Peripheral Interface
SQL	Structured Query Language
TIS	Tool Interface Standard
TORA	Temporally-Ordered Routing Algorithm
URL	Uniform Resource Locator
USB	Universal Serial Bus
W3C	World Wide Web Consortium
WPAN	Wireless Personal Area Network
WSDL	Web Services Description Language
WWW	World Wide Web
XML	Extensible Markup Language

Literaturverzeichnis

[1] Abrach, Hector A. T.; Bhatti, Shah; Carlson, James; Dai, Hui; Rose, Jeff; Sheth, Anmol; Shucker, Brian; Deng, Jing; Han, Richard: MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors. In: *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*. San Diego, CA, USA, September 2003, S. 50–59

[2] al., Ingo M.: *Service-orientierte Architekturen mit Web Services: Konzepte – Standards – Praxis*. 3. Aufl. Spektrum Akademischer Verlag, 2008. – ISBN 9783827419934

[3] Ausiello, Giorgio; Crescenzi, Pierluigi; Gambosi, Giorgio; Kann, Viggo; Marchietti-Spaccamela, Albertio; Protasi, Marco: *Complexity and Approximation: Combinatorial Optimization and Their Approximabillity Properties*. Springer Verlag, 1999. – ISBN 3–540–65431–3

[4] Beutel, Jan; Kasten, Oliver; Ringwald, Matthias: Poster abstract: BTnodes – a distributed platform for sensor nodes. In: *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*. New York, NY, USA : ACM, 2003. – ISBN 1–58113–707–9, S. 292–293

[5] Broch, Josh; Maltz, David A.; Johnson, David B.; Hu, Yih-Chun; Jetcheva, Jorjeta: A performance comparison of multi-hop wireless ad hoc network routing protocols. In: *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*. New York, NY, USA : ACM Press, 1998. – ISBN 1–58113–035–X, S. 85–97

[6] BTnode Project - ETH Zurich: *BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks*. `www.btnode.ethz.ch`, Abruf: April 2009

[7] Cape Breton University: *PAWS - Petroleum Applications of Wireless System*. `www.paws.cbu.ca`. Version: Februar 2009

[8] Clausen, Thomas; Jacquet, Pierre: *Optimized Link State Routing Protocol (OLSR)*. RFC 3626 (Experimental). `http://www.ietf.org/rfc/rfc3626.txt`. Version: Oktober 2003 (Request for Comments)

[9] Cornell University: *Cornell Database Group - Cougar*. `http://www.cs.cornell.edu/bigreddata/cougar/index.php`, Abruf: Mai 2009

[10] Corson, M. S.; Macker, Joseph: *Mobile Ad hoc Networking (MANET): Routing Protocol Performance Issues and Evaluation Considerations*. RFC 2501 (Informational). `http://www.ietf.org/rfc/rfc2501.txt`. Version: Januar 1999 (Request for Comments)

[11] Crossbow Technology, Inc.: *Crossbow Technology*. http://xbow.com, Abruf: April 2009

[12] Day, John D.; Zimmermann, Hubert: The OSI reference model. In: *Proceedings of the IEEE* 71 (1983), Nr. 12, 1334–1340. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1457043

[13] Defense Advanced Research Projects Agency: *DARPA*. http://www.darpa.mil, Abruf: April 2009

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

[15] Dollimore, Jean; Kindberg, Tim; Coulouris, George: *Distributed Systems: Concepts and Design (4th Edition) (International Computer Science Series)*. Addison Wesley, 2005. – ISBN 0321263545

[16] Dunkels, Adam: *The Contiki Operating System*. http://www.sics.se/contiki/, Abruf: Mai 2009

[17] Dunkels, Adam; Finne, Niclas; Eriksson, Joakim; Voigt, Thiemo: Run-time dynamic linking for reprogramming wireless sensor networks. In: *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*. New York, NY, USA : ACM Press, 2006. – ISBN 1595933433, S. 15–28

[18] Dunkels, Adam; Gronvall, Bjorn; Voigt, Thiemo: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. Washington, DC, USA : IEEE Computer Society, 2004, S. 455–462

[19] Fall, Kevin; Varadhan, Kannan: *The ns Manual*. The VINT Project - a collaboration between researchers, 1989–2001. http://www.isi.edu/nsnam/ns/doc/

[20] Fekete, Sándor P.; Fischer, Stefan; Kröller, Alexander; Pfisterer, Dennis: *Shawn. Simulator for sensor networks by the SwarmNet project*. 2004. – http://www.swarmnet.de/shawn

[21] Fielding, Roy T.; Gettys, Jim; Mogul, Jeffrey; Frystyk, Henrik; Masinter, Larry; Leach, Paul J.; Berners-Lee, Tim: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616 (Draft Standard). http://www.ietf.org/rfc/rfc2616.txt. Version: Juni 1999 (Request for Comments). – Updated by RFC 2817

[22] Frank, Christian; Römer, Kay: Distributed Facility Location Algorithms for Flexible Configuration of Wireless Sensor Networks. In: *Distributed Computing in Sensor Systems - Third IEEE International Conference, DCOSS 2007*. Santa Fe, NM, USA, Juni 2007. – ISBN 978

[23] Free Software Foundation, Inc: *GCC online documentation GNU Project Free Software Foundation (FSF)*. http://gcc.gnu.org/onlinedocs/, Abruf: Juni 2009

[24] Furuta, Takehiro; Sasaki, Mihiro; Ishizaki, Fumio; Suzuki, Atsuo; Miyazawa, Hajime: A new clustering algorithm using facility location theory for wireless sensor networks / Nanzan Academic Society Mathematical Sciences and Information Engineering. 2007. – Forschungsbericht

[25] Gay, David; Levis, Philip; von Behren, Robert; Welsh, Matt; Brewer, Eric; Culler, David: 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* Bd. 38. New York, NY, USA : ACM Press, Mai 2003. – ISBN 1581136625, S. 1–11

[26] Glombitza, Nils; Hellbrück, Horst; Fischer, Stefan: Evaluating Ad Hoc Networks with FRED. In: *The 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM Mobihoc 2008), Demo Session*, ACM, Mai 2008. – ISBN 978–1–60558–0073–9

[27] Glombitza, Nils; Lipphardt, Martin; Werner, Christian; Fischer, Stefan: Using Graphical Process Modeling for Realizing SOA Programming Paradigms in Sensor Networks. In: *Sixth Annual Conference on Wireless On demand Network Systems and Services*. Snowbird, Utah, USA, 2009, S. 61–68

[28] Haas, Zygmunt. J.; Halpern, Joseph. Y.; Li, Li: Gossip-based Ad Hoc Routing. In: *Networking, IEEE/ACM Transactions on* 14 (2006), Nr. 3, S. 479–491. http://dx.doi.org/10.1109/TNET.2006.876186. – DOI 10.1109/TNET.2006.876186

[29] Han, Chih-Chieh; Kumar, Ram; Shea, Roy; Kohler, Eddie; Srivastava, Mani: A dynamic operating system for sensor nodes. In: *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA : ACM, 2005. – ISBN 1–931971–31–5, S. 163–176

[30] Hartung, Carl; Han, Richard; Seielstad, Carl; Holbrook, Saxon: FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In: *MobiSys '06: Proceedings of the 4th international conference on Mobile systems, applications and services*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–195–3, S. 28–41

[31] Heinzelman, Wendi; Chandrakasan, Anantha; Balakrishnan, Hari: An Application-Specific Protocol Architecture for. In: *IEEE Transactions on Wireless Communications* 1 (2002), Nr. 4

[32] Hellbrück, Horst; Lipphardt, Martin; Pfisterer, Dennis; Ransom, Stefan; Fischer, Stefan: Praxiserfahrungen mit MarathonNet - Ein mobiles Sensornetz im Sport. In: *Praxis der Informationsverarbeitung und Kommunikation (PIK)* 29 (2006), Oktober - Dezenber, Nr. 4, S. 195–202

[33] Hellbrück, Horst; Xin, Hua; Lipphardt, Martin: Effective Movement Classification for Context Awareness in Medical Application Networking. In: *Proceedings of the IEEE International Conference on Communications (ICC'09)*. Dresden, Germany, Juni 2009

[34] Hensel, Tobias: *Entwicklung und Durchführung einer Online-Umfrage zu MarathonNet*, Technische Universität Carolo-Wilhelmina zu Braunschweig, Diplomarbeit, September 2005

[35] Hill, Jason; Szewczyk, Robert; Woo, Alec; Hollar, Seth; Culler, David; Pister, Kristofer: System architecture directions for networked sensors. In: *SIGPLAN Not.* 35 (2000), Nr. 11, S. 93–104. http://dx.doi.org/10.1145/356989.356998. – DOI 10.1145/356989.356998. – ISSN 0362–1340

[36] Hill, Jason L.; Culler, David E.: Mica: A Wireless Platform for Deeply Embedded Networks. In: *IEEE Micro* 22 (2002), Nr. 6, S. 12–24. http://dx.doi.org/10.1109/MM.2002.1134340. – DOI 10.1109/MM.2002.1134340. – ISSN 0272–1732

[37] Hill, Jason L.: *System architecture for wireless sensor networks*, University of California, Berkeley, Diss., 2003

[38] Hui, Jonathan W.; Culler, David: The dynamic behavior of a data dissemination protocol for network programming at scale. In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–879–2, S. 81–94

[39] ISO: *ISO - International Organization for Standardization*. www.iso.org/iso/home.htm, Abruf: Mai 2009

[40] Jacquet, Pierre; Mühlethaler, Paul; Clausen, Thomas; Laouiti, Anis; Qayyum, Amir; Viennot, Laurent: Optimized link state routing protocol for ad hoc networks. In: *Proceedings of the 5th IEEE Multi Topic Conference (INMIC 2001)*, 2001

[41] Johnson, David B.: Routing in Ad Hoc Networks of Mobile Hosts. In: *In Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 1994, S. 158–163

[42] Johnson, David B.; Hu, Yih-Chun; Maltz, David A.: *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4*. RFC 4728 (Experimental). http://www.ietf.org/rfc/rfc4728.txt. Version: Februar 2007 (Request for Comments)

[43] Johnson, David B.; Maltz, David A.: Dynamic Source Routing in Ad Hoc Wireless Networks. In: *Mobile Computing*, Kluwer Academic Publishers, 1996, S. 153–181

[44] Johnstone, Ian; Nicholson, James; Shehzad, Babar; Slipp, Jeff: Experiences from a wireless sensor network deployment in a petroleum environment. In: *IWCMC '07: Proceedings of the 2007 international conference on Wireless communications and mobile computing*. New York, NY, USA : ACM, August 2007. – ISBN 978–1–59593–695–0, S. 382–387

[45] Karl, Holger; Willig, Andreas: *Protocols and Architectures for Wireless Sensor Networks*. Wiley-Interscience, 2007. – ISBN 0470519231

[46] Karp, Brad; Kung, Hsiang-Tsung: GPSR: greedy perimeter stateless routing for wireless networks. In: *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*. New York, NY, USA : ACM Press, 2000. – ISBN 1581131976, S. 243–254

[47] Klensin, John: *Simple Mail Transfer Protocol*. RFC 2821 (Proposed Standard). `http://www.ietf.org/rfc/rfc2821.txt`. Version: April 2001 (Request for Comments)

[48] Kotz, David; Newport, Calvin; Elliott, Chip: The mistaken axioms of wireless-network research / Dartmouth College. 2003. – Forschungsbericht

[49] Krivitski, Denis; Wolff, Assaf S.: A Local Facility Location Algorithm for Large-Scale Distributed Systems. In: *Journal of Grid Computing* 5 (2007), Dezember, Nr. 4, S. 361–378

[50] Kurkowski, Stuart; Camp, Tracy; Colagrosso, Michael: MANET Simulation Studies: The Incredibles. In: *SIGMOBILE Mobile Computing and Communications Review* 9 (2005), Oktober, Nr. 4, S. 50–61. – ISSN 1559–1662

[51] Lamport, Leslie: Time, Clocks, and the Ordering of Events in a Distributed System. In: *Commun. ACM* 21 (1978), Nr. 7, S. 558–565. `http://dx.doi.org/10.1145/359545.359563`. – DOI 10.1145/359545.359563. – ISSN 0001–0782

[52] Laoutaris, N.; Smaragdakis, G.; Oikonomou, K.; Stavrakakis, I.; Bestavros, A.: Distributed Placement of Service Facilities in Large-Scale Networks. In: *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, 2007, S. 2144–2152

[53] Lee, Sung J.; Gerla, Mario: AODV-BR: Backup Routing in Ad hoc Networks. In: *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2000)*. Chicago, IL, Sepember 2000

[54] Lee, William C. Y.: *Mobile Communications Engineering: Theory and Applications*. 2. New York, NY, USA : McGraw-Hill Professional, 1997. – ISBN 1590611357

[55] Levis, Philip; Culler, David: 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*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–574–2, S. 85–95

[56] Levis, Philip; Madden, Samuel R.; Gay, David; Polastre, Joseph; Szewczyk, Robert; Woo, Alec; Brewer, Eric; Culler, David: The emergence of networking abstractions and techniques in TinyOS. In: *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA : USENIX Association, 2004, 1

[57] Levis, Philip; Madden, Samuel R.; Polastre, Joseph; Szewczyk, Robert; Whitehouse, Kamin; Woo, Alec; Gay, David; Hill, Jason L.; Welsh, Matt; Brewer, Eric.; Culler, David: TinyOS: An Operating System for Sensor Networks. In: *Ambient Intelligence* (2005), S. 115–148.

http://dx.doi.org/10.1007/3-540-27139-2_7. – DOI 10.1007/3–540–27139–2_7

[58] Li, Baochun; Wang, Karen H.: NonStop: Continuous Multimedia Streaming in Wireless Ad Hoc Networks with Node Mobility. In: *IEEE Journal on Selected Areas in Communications* 21 (2003), S. 1627–1641

[59] Li, Jinyang; Blake, Charles; Couto, Douglas S. J. D.; Lee, Hu I.; Morris, Robert: Capacity of Ad Hoc Wireless Networks. In: *MobiCom'01: Proceedings of the 7th ACM International Conference on Mobile Computing and Networking*. Rom, Italien, Juli 2000, S. 61–69

[60] Li, Shuoqi; Ying, Lin; Son, Sang H.; Stankovic, John A.; Wei, Yuan: Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In: *Telecommunication Systems* 26 (2004), S. 351–368

[61] Lipphardt, Martin; Glombitza, Nils; Neumann, Jana; Werner, Christian: A Service-oriented Operating System and an Application Development Infrastructure for Wireless Sensor Networks (Demo). In: Abdelzaher, Tarek F. (Hrsg.); Martonosi, Margaret (Hrsg.); Wolisz, Adam (Hrsg.): *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys 2009)*. Berkeley, California, USA : ACM, November 4 - 6 2009

[62] Lipphardt, Martin; Hellbrück, Horst; Wegener, Axel; Fischer, Stefan: GRAPE - Gradient based Routing for All PurposE. In: *Proceedings of the 4th IEEE International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*. Pisa, Italy, Oktober 2007, S. 1–6

[63] Lipphardt, Martin; Hellbrueck, Horst; Pfisterer, Dennis; Ransom, Stefan; Fischer, Stefan: Practical Experiences on Mobile Inter-Body-Area-Networking. In: *Proceedings of the Second International Conference on Body Area Networks (BodyNets'07)*, 2007

[64] Lipphardt, Martin; Neumann, Jana; Groppe, Sven; Werner, Christian: DySSCo - A protocol for Dynamic Self-organizing Service Coverage. In: Hummel, Karin A. (Hrsg.); Sterbenz, James P. G. (Hrsg.): *Proceedings of the 3rd International Workshop on Self-Organizing Systems (IWSOS 2008)* Bd. 5343. Wien, Österreich : Springer-Verlag, December 10 - 12 2008 (Lecture Notes in Computer Science (LNCS)). – ISBN 978–3–540–92156–1, S. 109–120

[65] Lipphardt, Martin; Neumann, Jana; Hoeller, Nils; Reinke, Christoph; Groppe, Sven; Linnemann, Volker; Werner, Christian: XML und SOA als Wegbereiter für Sensornetze in der Praxis. In: *Praxis der Informationsverarbeitung und Kommunikation (PIK)* 31 (2008), Juli - September, Nr. 3, S. 146–152

[66] Lipphardt, Martin; Neumann, Jana; Werner, Christian: Self-organizing Service Distribution (Demo). In: Abdelzaher, Tarek F. (Hrsg.); Martonosi, Margaret (Hrsg.); Wolisz, Adam (Hrsg.): *Proceedings of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys 2008)*. Raleigh, NC, USA : ACM, November 5 - 7 2008. – ISBN 978–1–59593–990–6, S. 389–390

[67] Liu, Jie; Cheung, Patrick; Guibas, Leonidas; Zhao, Feng: Apply Geometric Duality to Energy Efficient Non-Local Phenomenon Awareness using Sensor Networks. In: *IEEE Wireless Communication Magazine* 11 (2004), Oktober, Nr. 5, S. 62–68

[68] Lorincz, Konrad; Malan, David J.; Fulford-Jones, Thaddeus R. F.; Nawoj, Alan; Clavel, Antony; Shnayder, Victor; Mainland, Geoffrey; Welsh, Matt; Moulton, Steve: Sensor Networks for Emergency Response: Challenges and Opportunities. In: *IEEE Pervasive Computing* 3 (2004), Oktober - Dezember, Nr. 4, S. 16–23. http://dx.doi.org/10.1109/MPRV.2004.18. – DOI 10.1109/MPRV.2004.18. – ISSN 1536–1268

[69] Lymberopoulos, Dimitrios; Savvides, Andreas: XYZ: a motion-enabled, power aware sensor node platform for distributed sensor network applications. In: *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*. Piscataway, NJ, USA : IEEE Press, 2005. – ISBN 0–7803–9202–7, S. 63

[70] Madden, Samuel R.; Franklin, Michael J.; Hellerstein, Joseph M.; Hong, Wei: TinyDB: an acquisitional query processing system for sensor networks. In: *ACM Trans. Database Syst.* 30 (2005), Nr. 1, S. 122–173. http://dx.doi.org/10.1145/1061318.1061322. – DOI 10.1145/1061318.1061322. – ISSN 0362–5915

[71] Mainwaring, Alan; Culler, David; Polastre, Joseph; Szewczyk, Robert; Anderson, John: Wireless sensor networks for habitat monitoring. In: *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*. New York, NY, USA : ACM Press, September 2002. – ISBN 1–58113–589–0, S. 88–97

[72] Malan, David; Fulford-jones, Thaddeus; Welsh, Matt; Moulton, Steve: CodeBlue: An ad hoc sensor network infrastructure for emergency medical care. In: *Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks*. Washington, DC, USA : IEEE Computer Society, April 2004

[73] MANTIS Group at Colorado University, Boulder: *MANTIS : Homepage*. http://mantis.cs.colorado.edu/, Abruf: Mai 2009

[74] Moscibroda, Thomas; Wattenhofer, Roger: Facility location: distributed approximation. In: *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–994–2, S. 108–117

[75] Mottola, Luca; Picco, Gian P.: Programming wireless sensor networks with logical neighborhoods: a road tunnel use case. In: *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–763–6, S. 393–394

[76] Nasipuri, Asis; Das, Samir R.: On-demand multipath routing for mobile ad hoc networks. In: *Eighth International Conference on Computer Communications and Networks*, 1999, S. 64–70

[77] OASIS WS-BPEL Technical Committee: *Webservices – Business Process Execution Language Version 2.0*. 2005

[78] Object Management Group: *Catalog of OMG CORBA/IIOP Specifications*. Version: April 2009. `http://www.omg.org/technology/documents/corba_spec_catalog.htm`, Abruf: August 2009

[79] Organization for the Advancement of Structured Information Standards (OASIS): *SOA-RM Wiki*. `http://wiki.oasis-open.org/soa-rm/FrontPage`, Abruf: März 2009

[80] Organization for the Advancement of Structured Information Standards (OASIS): *Reference Model for Service Oriented Architecture 1.0 - OASIS Standard, 12 October 2006*. Version: Oktober 2006. `http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf`, Abruf: März 2009

[81] Park, Vincent D.; Corson, M. S.: A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. In: *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE* Bd. 3, 1997

[82] Perkins, Charles E.; Belding-Royer, Elizabeth M.; Das, Samir R.: *Ad hoc On-Demand Distance Vector (AODV) Routing*. RFC 3561 (Experimental). `http://www.ietf.org/rfc/rfc3561.txt`. Version: Juli 2003 (Request for Comments)

[83] Perkins, Charles E.; Bhagwat, Pravin: Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In: *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, 1994, S. 234–244

[84] Perkins, Charles E.; Royer, Elizabeth M.: Ad-hoc On-Demand Distance Vector Routing. In: *In Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, 1997, S. 90–100

[85] Pfisterer, Dennis; Lipphardt, Martin; Buschmann, Carsten; Hellbrueck, Horst; Fischer, Stefan; Sauselin, Jan H.: MarathonNet: Adding value to large scale sport events - A connectivity analysis. In: *InterSense '06: Proceedings of the International Conference on Integrated Internet Ad hoc and Sensor Networks*, 2006

[86] Polar Electro Oy: *Polar Electro - Deutschland*. `www.polar-deutschland.de/`, Abruf: April 2009

[87] Postel, Jon; Reynolds, Joyce K.: *File Transfer Protocol*. RFC 959 (Standard). `http://www.ietf.org/rfc/rfc959.txt`. Version: Oktober 1985 (Request for Comments). – Updated by RFCs 2228, 2640, 2773, 3659

[88] Read, Ronald C.; Wilson, Robin J.: *An atlas of graphs*. New York : The Clarendon Press Oxford University Press, 1998. – ISBN 0–19–853289–X

[89] Reinke, Christoph; Hoeller, Nils; Lipphardt, Martin; Neumann, Jana; Groppe, Sven; Linnemann, Volker: Integrating Standardized Transaction Protocols in Service Oriented Wireless Sensor Networks. In: *Proceedings of the 24th ACM Symposium on Applied Computing (ACM SAC 2009)*. Honolulu, Hawaii, USA : ACM, March 8 - 12 2009. – ISBN 978–1–60558–166–8, S. 2202–2203

[90] Royer, Elizabeth M.; Toh, Chai-Keong: A Review of Current Routing Protocols for Ad-Hoc Mobile Wireless Networks. In: *IEEE Personal Communications* (1999), April, S. 46

[91] Sailhan, Francoise; Issarny, Valerie: Scalable Service Discovery for MANET. In: *PERCOM '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications*. Washington, DC, USA : IEEE Computer Society, 2005. – ISBN 0–7695–2299–8, S. 235–244

[92] ScatterWeb GmbH: *ScatterWeb – wireless network solutions*. `http://www.scatterweb.com/`, Abruf: Mai 2009

[93] Schmidtke, Heinz: *Handbuch der Ergonomie: HdE, mit ergonomischen Konstruktionsrichtlinien und Methoden*. 2. Auflage. Hanser Verlag, 1998

[94] Sedgewick, Robert: *Algorithmen in Java*. Pearson Studium, 2003. – ISBN 978–3–8273–7072–3

[95] Shnayder, Victor; Chen, Bor R.; Lorincz, Konrad; Thaddeus; Welsh, Matt: Sensor Networks for Medical Care / Harvard University. 2005 (TR-08-05). – Forschungsbericht

[96] Shukla, SK; Rosenkrantz, DJ; Ravi, SS: Observations on self-stabilizing graph algorithms for anonymous networks. In: *Proceedings of the Second Workshop on Self-Stabilizing Systems*, 1995, 7.1–7.15

[97] Sun Microsystems: *Remote Method Invocation Home*. `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp`, Abruf: August 2009

[98] Sun Microsystems: *RPC: Remote Procedure Call Protocol specification: Version 2*. RFC 1057 (Informational). `http://www.ietf.org/rfc/rfc1057.txt`. Version: Juni 1988 (Request for Comments)

[99] Szewczyk, Robert; Mainwaring, Alan; Polastre, Joseph; Anderson, John; Culler, David: An analysis of a large scale habitat monitoring application. In: *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–879–2, S. 214–226

[100] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. 2., überarb. A. Pearson Studium, 2002. – ISBN 3827370191

[101] Tanenbaum, Andrew S.: *Computernetzwerke*. Pearson Studium, 2003. – ISBN 3827370469

[102] Tanenbaum, Andrew S.; Steen, Maarten van: *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, 2006. – ISBN 0132392275

[103] Tel, Gerard: *Introduction to Distributed Algorithms.* 2. Cambridge University Press, 2001. – ISBN 0521794838

[104] Texas Instruments Incorporated: *Analog Technologies, Semiconductors, Digital Signal Processing - Texas Instruments.* http://www.ti.com/, Abruf: Mai 2009

[105] The ACM Special Interest Group on Mobility of Systems, Users, Data, And Computing (ACM SIGMOBILE: *MobiHoc: The ACM International Symposium on Mobile Ad Hoc Networking and Computing.* http://www.sigmobile.org/mobihoc/, Abruf: August 2009

[106] TIS Commitee: *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2.* Mai 1995

[107] Tseng, Yu-Chee; Ni, Sze-Yao; Chen, Yuh-Shyan; Sheu, Jang-Ping: The broadcast storm problem in a mobile ad hoc network. In: *Wirel. Netw.* 8 (2002), Nr. 2/3, S. 153–167. http://dx.doi.org/10.1023/A:1013763825347. – DOI 10.1023/A:1013763825347. – ISSN 1022–0038

[108] Tsirigos, Aristotelis; Haas, Zygmunt J.; Member, Senior: Analysis of multipath routing-part I: The effect on the packet delivery ratio. In: *IEEE Transactions on Wireless Communications* 3 (2004), S. 138–146

[109] Undersea Warfare U.S. Submarine Forces: *SOSUS The Secret Weapon of Undersea Surveillance.* http://www.navy.mil/navydata/cno/n87/usw/issue_25/sosus.htm, Abruf: Juli 2008

[110] University of California, Bekreley: *TinyOS Community Forumm - An open source OS for the networked sensor regime.* http://www.tinyos.net/, Abruf: Mai 2009

[111] University of California, Berkeley: *SMART DUST - Autonomous sensing and communication in a cubic millimeter.* http://robotics.eecs.berkeley.edu/~pister/SmartDust/, Abruf: Juli 2008

[112] University of California, Berkeley: *TinyDB: A Declarative Database for Sensor Networks.* http://telegraph.cs.berkeley.edu/tinydb/, Abruf: Mai 2009

[113] University of California, Los Angeles - Networked & Embedded Systems Laboratory (NESL): *SOS 2.x Home Page.* https://projects.nesl.ucla.edu/public/sos-2x/doc/, Abruf: Mai 2009

[114] Voigt, Thiemo; Tsiftes, Nicolas; He, Zhitao: Remote water monitoring with sensor networking technology. In: *ERCIM News* (2009), Nr. 76, 0. http://ercim-news.ercim.org/content/view/510/705/

[115] Walletex Microelectronics Ltd: *Wallet MP3 - a credit-card-sized MP3 player in your wallet.* www.walletex.com/gp.asp?gpid=117, Abruf: Juli 2008

[116] Weigle, Michele C.: Improving confidence in network simulations. In: Perrone, L. F. (Hrsg.); Lawson, Barry G. (Hrsg.); Liu, Jason (Hrsg.); Wieland, Frederick P. (Hrsg.): *Proceedings of the 38th conference on Winter simulation*, Winter Simulation Conference, Dezember 2006. – ISBN 1–4244–0501–7, S. 2188–2194

[117] Weiser, Mark: The Computer for the 21st Century. In: *Scientific American* (1991), September, 94-10. http://www.ubiq.com/hypertext/weiser/UbiHome.html

[118] Whitehouse, Kamin; Sharp, Cory; Brewer, Eric; Culler, David: Hood: a neighborhood abstraction for sensor networks. In: *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, ACM, 2004. – ISBN 1–58113–793–1, S. 99–110

[119] World Wide Web Consortium (W3C): *World Wide Web Consortium - Web Standards*. http://www.w3.org, Abruf: August 2008

[120] World Wide Web Consortium (W3C): *Web Services Description Language (WSDL) 1.1*. Version: März 2001. http://www.w3.org/TR/wsdl, Abruf: Juli 2009

[121] World Wide Web Consortium (W3C): *Recommendation: SOAP Version 1.2 Part 1: Messaging Framework*. Version: Juni 2003. http://www.w3.org/TR/soap12-part1/, Abruf: Juli 2009

[122] World Wide Web Consortium (W3C): *Recommendation: SOAP Version 1.2 Part 2: Adjuncts*. Version: Juni 2003. http://www.w3.org/TR/soap12-part2/, Abruf: Juli 2009

[123] World Wide Web Consortium (W3C): *Extensible Markup Language (XML)*. Version: April 2009. http://www.w3.org/XML/, Abruf: Juli 2009

[124] World Wide Web Consortium (W3C) Working Group: *Web Services Architecture - W3C Working Group Note 11 February 2004*. Version: Februar 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, Abruf: August 2008

[125] Yao, Yong; Gehrke, Johannes: The cougar approach to in-network query processing in sensor networks. In: *SIGMOD Rec.* 31 (2002), Nr. 3, S. 9–18. http://dx.doi.org/10.1145/601858.601861. – DOI 10.1145/601858.601861. – ISSN 0163–5808

[126] Ye, Fan; Zhong, Gary; Lu, Songwu; Zhang, Lixia: GRAdient broadcast: a robust data delivery protocol for large scale sensor networks. In: *Wireless Networks* 11 (2005), Nr. 3, S. 285–298. http://dx.doi.org/10.1007/s11276-005-6612-9. – DOI 10.1007/s11276–005–6612–9. – ISSN 1022–0038

[127] Yick, Jennifer; Mukherjee, Biswanath; Ghosal, Dipak: Wireless sensor network survey. In: *Comput. Netw.* 52 (2008), Nr. 12, S. 2292–2330. http://dx.doi.org/http://dx.doi.org/10.1016/j.comnet.2008.04.002. – DOI http://dx.doi.org/10.1016/j.comnet.2008.04.002. – ISSN 1389–1286

[128] Yoon, J.; Liu, M.; Noble, B.: Random waypoint considered harmful. In: *Proceedings of the 22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* Bd. 2, 2003, S. 1312–1321

[129] Zeng, Xiang; Bagrodia, Rajive; Gerla, Mario: GloMoSim: a library for parallel simulation of large-scale wireless networks. In: *Parallel and Distributed Simulation (PADS '98), 12th Workshop.* Washington, DC, USA : IEEE Computer Society, 1998. – ISBN 0–8186–8457–7, S. 154–161

[130] Zhou, Gang; He, Tian; Krishnamurthy, Sudha; Stankovic, John A.: Impact of radio irregularity on wireless sensor networks. In: *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services.* New York, NY, USA : ACM, 2004. – ISBN 1–58113–793–1, S. 125–138