Aus dem Institut für Telematik
der Universität zu Lübeck

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

Verteilte Protokollstapel für drahtlose Sensornetze

Inauguraldissertation
zur
Erlangung der Doktorwürde
der Universität zu Lübeck
Aus der Sektion Informatik/Technik

Vorgelegt von

Herrn Dipl.-Inf. Peter Rothenpieler

aus Hamburg

Lübeck, 2014

Erster Berichterstatter:	Prof. Dr. rer. nat. Stefan Fischer
Zweiter Berichterstatter:	Prof. Dr. rer. nat. Volker Linnemann
Tag der mündlichen Prüfung:	23. April 2014

Zum Druck genehmigt.

Lübeck, den 25. April 2014

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich bei der Bearbeitung dieser Arbeit unterstützt und begleitet haben. An erster Stelle sei hier mein Doktorvater genannt, dem ich diese Möglichkeit zu verdanken habe und von dem ich in den letzten Jahren viel lernen konnte. Ich danke Herrn Prof. Dr. Stefan Fischer ganz herzlich dafür, dass ich die Anleitung, das Vertrauen und die Freiräume genießen durfte, ohne die meine wissenschaftliche Arbeit nicht möglich gewesen wäre.

Die für mich persönlich wichtigsten Menschen im Leben sind meine Verlobte Nadine, meine Eltern und mein Bruder Mathias: Vielen Dank für eure Unterstützung und die vielen schönen Momente außerhalb der Forschung.

Ich habe meine Zeit am Institut für Telematik sehr genossen, was neben den spannenden Forschungsthemen zu einem großen Teil an den Menschen lag, mit denen ich zusammenarbeiten durfte. Es hat Spaß gemacht, mit Euch allen zu arbeiten!

Kurzfassung

Drahtlose Sensornetze ermöglichen es, physikalische Phänomene mittels eines Netzwerks aus Sensoren über einen beliebigen Zeitraum zu beobachten. Die auf diese Weise gesammelten Messwerte können dazu verwendet werden, ein Modell der Realität zu erstellen, das zur Analyse, Steuerung und Optimierung von Prozessen verwendet werden kann. In den letzten Jahren haben sich Sensornetze von maßgeschneiderten und problemspezifischen Lösungen zu standardisierten Hard- und Softwareplattformen entwickelt, die durch die Integration von IPv6 zu einem wesentlichen Bestandteil des Internets der Dinge geworden sind.

Die Verwendung von standardisieren Protokollen ermöglicht die Interoperabilität von Sensornetzen unterschiedlicher Hersteller und erlaubt die nahtlose Integration in bestehende Netzwerke, wie z.B. das Internet. Auf der anderen Seite sind die Ressourcenanforderungen dieser Protokolle im Vergleich zu den zuvor eingesetzten maßgeschneiderten Protokollen höher, was den Einsatz dieser Protokolle auf den stärker ressourcenbeschränkten Sensorknoten einschränkt oder unmöglich macht.

Der Mangel an Ressourcen ist ein typisches Problem in drahtlosen Sensornetzen. Ein einzelner Sensorknoten verfügt häufig nicht über die benötigten Ressourcen, um eine vorgegebene Aufgabe zu erfüllen. Aus diesem Grund bedient sich eine Vielzahl von Lösungsansätzen der Kooperation benachbarter Sensorknoten, die ihre Ressourcen gemeinsam zur Lösung einer Aufgabe verwenden.

Die vorliegende Arbeit stellt ein Verfahren vor, das die Bildung eines verteilten Protokollstapels durch die Kooperation benachbarter Sensorknoten ermöglicht. Dieser verteilte Protokollstapel ermöglicht es, alle Sensorknoten über die verwendeten Protokolle, wie z.B. IPv6, nahtlos in das Netzwerk zu integrieren, ohne dass die hierfür benötigten Protokolle auf allen Sensorknoten implementiert werden müssen. Stattdessen muss von den Sensorknoten nur eine Untermenge der Protokolle implementiert werden, während die fehlenden Protokolle mittels Remote-Procedure-Calls von den benachbarten Sensorknoten mitbenutzt werden können.

In dieser Arbeit wird ein Protokoll beschrieben, das diese Vorgehensweise ermöglicht. Neben einer Beschreibung der Protokollmechanismen wird hierbei auf einige Details der Implementierung eingegangen. Außerdem wird eine Evaluation des Protokolls sowohl in einer Umgebung mit Single-Hop-Kommunikation unter Laborbedingungen als auch in einer Umgebung mit Multi-Hop-Kommunikation anhand eines Anwendungsbeispiels durchgeführt.

Abstract

Wireless Sensor Networks enable the monitoring of physical conditions over long periods of time using a wireless network of distributed sensors. The gathered sensor data allows the creation of a virtual model of reality that can be used to analyse, control or optimize processes in the real world. Over the last few years, Wireless Sensor Networks have evolved from proprietary, custom tailored and problem-specific solutions to standardised hard- and software platforms, which, through the introduction of IPv6, have become an important part of the Internet of Things.

The use of standardised protocols allows interoperability between sensor networks from different manufacturers as well as their seamless integration into existing networks, such as the Internet. On the other hand, the resource requirements of these protocols are considerably higher than previously used custom tailored solutions. This limits and may even prohibit the use of such protocols on resource constrained devices.

The lack of resources is a typical problem for Wireless Sensor Networks. A single sensor node is typically not equipped with enough resources to fulfil a given task. A common solution for this problem is the use of cooperation between neighbouring nodes, which share their resources to solve complex problems.

The approach proposed in this thesis coordinates multiple neighbouring nodes to form a distributed protocol stack, whereby each node implements only a subset of the required communication protocols and utilizes missing protocol functionality from its surrounding nodes using Remote Procedure Calls. This distributed stack allows for the seamless integration of all nodes into the wireless network using communication protocols that may be too resource intensive for any single node to implement.

This thesis provides a detailed description of the protocol that allows the creation of these distributed protocol stacks. In addition to a description of the protocol mechanisms, this thesis includes some implementation remarks and an evaluation of the protocol. The protocol evaluation is performed in a one-hop environment under laboratory conditions as well as in a multi-hop environment using a real world application scenario.

Inhaltsverzeichnis

1. Einleitung

Die vorliegende Arbeit beschäftigt sich mit der Integration von drahtlosen Sensornetzen in das Internet der Dinge. Drahtlose Sensornetze sind ressourcenbeschränkte Kleinstcomputer, die es erlauben, physikalische Größen mit beliebiger räumlicher und zeitlicher Auflösung sensorisch zu erfassen. Die auf diese Weise gesammelten Sensorwerte können zur Modellierung der Realität, zur Optimierung von Prozessen oder zur Steuerung von Aktoren verwendet werden. Ein Anwendungsbeispiel für drahtlose Sensornetze ist die Gebäudeautomation, bei der z.B. die Temperatur, Luftfeuchtigkeit und Helligkeit in den einzelnen Räumen eines Bürogebäudes überwacht werden können. Auf der Grundlage dieser Daten kann dann eine bedarfsgerechte Steuerung der Heizungs- und Klimaanlage sowie der Beleuchtung erfolgen, die einerseits an den Tagesverlauf und die Jahreszeit und andererseits an die Arbeitszeiten und Anwesenheit der Mitarbeiter angepasst werden können.

Drahtlose Sensornetze haben in der Vergangenheit häufig proprietäre Kommunikationsprotokolle verwendet, um Informationen im Netzwerk auszutauschen. Die Anbindung an externe Computersysteme erfolgte in diesen Fällen in der Regel an einer Basisstation, welche die Protokollkonvertierung zwischen den proprietären Sensornetzprotokollen und den standardisierten Protokollen im Internet durchführte. In den vergangenen Jahren wurden die proprietären Protokolle im Sensornetz zunehmend durch die standardisierten Protokolle des TCP/IP-Referenzmodells ausgetauscht, um eine nahtlose Anbindung des Sensornetzes an das Internet zu ermöglichen. Auf diese Weise haben sich Sensornetze zu einem wesentlichen Bestandteil des Internet der Dinge entwickelt.

Das Internet der Dinge unterscheidet sich vom klassischen Internet dadurch, dass nicht länger nur Menschen auf Informationen zugreifen, die von anderen Menschen erzeugt wurden, sondern dass auch Maschinen selbständig Informationen erzeugen und austauschen können [1, 2]. Diese Maschinen dienen dazu, Dinge der realen Welt in der virtuellen Welt des Internet zu repräsentieren, um auf diese Weise Informationen über diese Dinge, z.B. in der Form von Sensorwerten, in Echtzeit von überall auf der Welt verfügbar zu machen. Eine der in diesem Zusammenhang bereits in den 1990er Jahre entwickelte Zukunftsvision wird als Smart Dust bezeichnet, bei der Sensorknoten durch die voranschreitende Miniaturisierung auf die Größe eines Staubkorns verkleinert werden sollen. Erste Sensorknoten in der Größe eines Reiskorns ($16\,\mathrm{mm}^3$, siehe [3]) wurden um die Jahrtausendwende von Forschern an der University of Berkeley in Kalifornien präsentiert. Um die Zukunftsvision des Smart Dust zu verwirklichen, müssen Sensorknoten folglich von Jahr zu Jahr immer kleiner und günstiger werden. Dies steht im Gegensatz zu klassischen Computern wie z.B. Desktop-PCs und

Laptops, die bei gleichbleibender Größe und gleichbleibendem Preis über eine immer höhere Rechenkapazität und größeren Speicherplatz verfügen.

1.1. Motivation

Um die Vision des Smart Dust in Zusammenhang mit dem Internet der Dinge realisieren zu können, müssen noch zahlreiche Herausforderungen überwunden werden. Sollen im Rahmen des Smart Dust Tausende oder Millionen Sensorknoten zur Überwachung von Objekten in der Realität verwendet werden, müssen diese Sensorknoten immer kleiner und vor allem jedoch immer günstiger werden. Dies liegt einerseits an den Anschaffungskosten eines derart großen Sensornetzes und andererseits an dem Umstand, dass Sensorknoten von der Größe eines Staubkorns in vielen Anwendungsfällen als Wegwerfartikel mit begrenzter Lebensdauer betrachtet werden müssen. Aufgrund dieser Eigenschaft müssen Sensornetze redundant ausgelegt sein und defekte Sensorknoten in regelmäßigen Abständen ausgetauscht werden. Auch der Energieverbrauch spielt in diesem Zusammenhang eine wesentliche Rolle. Je niedriger der Energieverbrauch, desto länger können Sensorknoten im Batteriebetrieb ihrer Aufgabe nachgehen, ohne dass diese neu geladen oder ausgetauscht werden muss. Im Gegensatz zu klassischen Computern stehen bei Sensorknoten folglich nicht die Rechenkapazität oder der Speicherplatz, sondern vielmehr der Preis, die Baugröße und der Energieverbrauch im Vordergrund [4–6]. Dies spiegelt sich auch in der Entwicklung der Hardwareplattformen für Sensornetze wieder, von denen einige in [7] zusammengestellt sind. Die in der Arbeit von [7] vorgestellten Plattformen haben sich von universellen Mikrocontrollern mit viel Speicher und hohem Energieverbrauch zu spezialisierten Sensorknoten mit wenig Speicher und niedrigem Energieverbrauch entwickelt.

Die Anbindung von Sensornetzen an das Internet der Dinge erfolgt unter anderem aufgrund der großen Anzahl an benötigten Adressen mittels IPv6 [8] im Zusammenspiel mit dem 6LoWPAN-Protokoll [9]. In den letzten Jahren sind aus diesem Grund eine Vielzahl von IPv6-Implementierungen für verschiedene Sensornetz-Betriebssysteme entstanden. Diese Implementierungen haben gezeigt, dass der Betrieb von IPv6 in Sensornetzen für unterschiedliche Anwendungsgebiete möglich ist. Im Gegensatz zu den proprietären Protokollen, die in der Form von maßgeschneiderten Lösungen auf die jeweilige Hardwareplattform und das Anwendungsgebiet angepasst werden konnten, sind die Anforderungen eines IPv6-Protokollstapels an den Programmspeicher der Sensorknoten jedoch deutlich höher. Diese Erfahrung musste auch der Autor dieser Arbeit machen, als er am Institut für Telematik in Zusammenarbeit mit der Firma coalesenses die Betreuung einer Diplomarbeit übernahm, welche die Implementierung von IPv6 und 6LoWPAN für die iSense-Sensorknotenplattform zum Ziel hatte. Ein Ergebnis dieser Arbeit war, dass der zur Verfügung stehende Programmspeicher einiger Hardwareplattformen für den Betrieb von IPv6 nicht ausreicht. Dieses Problem betrifft viele Plattformen und Betriebssysteme für drahtlose Sensornetze.

In Abhängigkeit von dem jeweiligen Anwendungsgebiet können neben IPv6 und 6LoWPAN noch weitere Protokolle auf der Anwendungsschicht oder zur Integration von Sicherheitsmechanismen wie z.B. IPsec oder TLS für den Betrieb eines Sensornetzes notwendig sein, wodurch der Bedarf an Programmspeicher weiter erhöht wird. Dies kann durch die Erhöhung des Programmspeichers ausgeglichen werden, was aus finanzieller Sicht jedoch auch eine Erhöhung des Preises jedes einzelnen Sensorknotens bedeutet. Zusätzlich konkurrieren jedoch noch die Energieversorgung, die Funkschnittstelle, die Qualität und Anzahl der Sensoren sowie viele weitere Komponenten um die zur Verfügung stehenden finanziellen Mittel.

Die Problematik, dass die auf einem einzelnen Sensorknoten zur Verfügung stehenden Ressourcen nicht zur Erfüllung einer Aufgabe ausreichen, ist typisch für drahtlose Sensornetze. Eine mögliche Lösung stellt hierbei die Kooperation benachbarter Sensorknoten dar, die ihre Ressourcen miteinander teilen und somit komplexe Aufgaben lösen können, die von einem einzelnen Sensorknoten nicht bewältigt werden könnten. Während die Kooperation zwischen benachbarten Sensorknoten bereits für eine Vielzahl von Problemstellungen und Anwendungsszenarien erfolgreich verwendet wurden (siehe Abschnitt 2.1.5), wurde dies bisher bei der Bildung eines Protokollstapels nicht untersucht.

1.2. Wissenschaftliche Beiträge und Aufbau der Arbeit

Die vorliegende Arbeit beschreibt, wie durch die Kooperation von benachbarten Sensorknoten in einem Sensornetz ein verteilter Protokollstapel gebildet werden kann, der die nahtlose Verbindung aller Sensorknoten untereinander sowie mit dem Internet ermöglicht, ohne dass alle Sensorknoten den vollständigen Protokollstapel implementieren müssen. Auf diese Weise kann der auf den einzelnen Sensorknoten für den Protokollstapel benötigte Programmspeicher reduziert werden. Hierdurch wird z.B. der Einsatz von IPv6 auch auf stärker ressourcenbeschränkten Sensorknoten ermöglicht. Die wissenschaftlichen Beiträge dieser Arbeit lassen sich wie folgt zusammenfassen:

- Ein neues Verfahren, das die Verteilung der Implementierung eines Protokollstapels auf mehrere Sensorknoten und somit die Anbindung von stärker ressourcenbeschränkten Geräten an das Internet erlaubt
- Die Beschreibung des DPS-Protokolls, welches die Umsetzung dieses Verfahrens ermöglicht und das sich im Netzwerk selbständig zur Laufzeit organisiert und auf Veränderungen der Topologie reagiert
- Sicherheitsmechanismen, die zur Gewährleistung der Integrität und Authentizität der versendeten Nachricht dienen und Schutz vor Angriffen bieten
- Eine Implementierung dieses Protokolls für den effizienten Einsatz der verteilten Protokollstapel. Dies beinhaltet unterschiedliche Mechanismen zur Optimierung des Speicherverbrauchs, der Latenz und des Datendurchsatzes

- Die umfassende Evaluation des Protokolls anhand von Experimenten, die sowohl die Auswirkungen auf die Single-Hop-Kommunikation als auch den Einsatz in einem Anwendungsszenario mit Multi-Hop-Kommunikation zeigen
- Richtlinien, die zur Anpassung des DPS-Protokolls an das jeweilige Anwendungsszenario und für die Konfiguration der optionalen Protokollbestandteile verwendet werden können

Das nachfolgende Kapitel beschäftigt sich zunächst mit einigen Grundlagen, die zum Verständnis dieser Arbeit hilfreich sind. Hierbei wird vor allem auf das Themengebiet der drahtlosen Sensornetze, deren Hard- und Softwareplattformen, einige Anwendungsgebiete sowie Beispiele zur Kooperation in Sensornetzen eingegangen. Außerdem werden einige Grundlagen zum Themengebiet der Schichtenmodelle präsentiert, wobei ein besonderer Schwerpunkt auf das TCP/IP-Referenzmodell und einige Sensornetz-spezifische Schichtenmodelle gesetzt wird.

Die Ausgangslage und Funktionsweise des in dieser Arbeit konzipierte Protokoll zur Bildung von verteilten Protokollstapeln (kurz DPS für Distributed Protocol Stacks) wird in Kapitel 3 beschrieben. wobei zu Beginn dieses Kapitels die bereits oben skizzierte Motivation durch Daten und Beispiele verdeutlicht wird. Anschließend wird das Konzept der verteilten Protokollstapel genauer vorgestellt und mit alternativen Lösungsansätzen und verwandten Arbeiten verglichen.

Kapitel 4 beinhaltet eine Beschreibung des Protokolls und des verwendeten Nachrichtenformats, das die Benutzung von Protokoll-Implementierungen auf benachbarten Sensorknoten erlaubt. Das Protokoll umfasst die folgenden Bestandteile:

- Mechanismen zum Suchen von und Verbinden mit Protokoll-Implementierungen auf benachbarten Sensorknoten
- Filtermöglichkeiten, so dass gezielt nach Protokoll-Implementierungen von Sensorknoten gesucht werden kann, die bestimmte Eigenschaften erfüllen, wie z.B. die Hardwareplattform oder der verbleibende Batterieladestand
- Mechanismen zur Überwachung existierender Verbindungen, um auf Veränderungen der Netzwerktopologie oder auf Knotenausfall reagieren zu können
- Sicherheitsmechanismen, welche die Integrität und Authentizität der versendeten Nachrichten gewährleisten
- Erweiterungen zur zustandslosen Kompression der Protokoll-Header und des Nachrichteninhalts zur Verbesserung der Latenz und des Datendurchsatzes

Im Anschluss wird in Kapitel 5 die Implementierung des Protokolls und seiner Bestandteile beschrieben, wobei neben dem Protokollablauf die Sicherheitsmechanismen und die verwendeten Mechanismen zur Kompression des Nachrichten-Headers und -Inhalts genau beschrieben werden. Den Abschluss von Kapitel 5

bilden zwei Beispiele, die zur Demonstration des in dieser Arbeit entwickelten Konzepts dienen. Das erste Beispiel verwendet das DPS-Protokoll, um das IPv6-Protokoll auf der Vermittlungsschicht zwischen benachbarten Sensorknoten zu teilen. Das zweite Beispiel integriert ein zentralisiertes Routing-Protokoll über die Infrastruktur, die durch ein Testbed zur Verfügung gestellt wird.

Die im vorangegangenen Kapitel vorgestellten Beispiele werden in Kapitel 6 zur Single-Hop-Evaluation des Protokolls verwendet, wobei neben der Quellcodegröße und dem Speicherverbrauch vor allem auf die Auswirkungen auf die Latenz und den Datendurchsatz eingegangen wird. Die Zusammenfassung dieses Kapitels beinhaltet Richtlinien zur Konfiguration des Protokolls und für den Einsatz der Mechanismen. Anschließend wird in Kapitel 7 eine Evaluation des DPS-Protokolls in einem Multi-Hop-Szenario mittels eines Anwendungsfalls aus der Gebäudeautomation durchgeführt.

Kapitel 8 bildet den Abschluss dieser Arbeit und beinhaltet eine Zusammenfassung sowie einen Ausblick auf zukünftige Arbeiten.

Zusammenarbeit mit anderen Forschern

Die in dieser Arbeit vorgestellten Ergebnisse wären ohne die Zusammenarbeit mit anderen Forschern nicht möglich gewesen. Die Idee der verteilten Protokollstapel und damit die Motivation hinter dieser Arbeit ist durch die Probleme entstanden, die während des Projekts FleGSens sowie der Implementierung des iSense-IPv6-Stacks im Rahmen einer Diplomarbeit, die vom Autor dieser Arbeit betreut wurde, entstanden. In beiden Projekten reichte der auf den verwendeten Sensorknoten verfügbare Speicherplatz nicht, um alle Protokolle auf den Sensorknoten auszuführen.

Die Sicherheitsmechanismen, die vom DPS-Protokoll verwendet werden, wurden in Zusammenarbeit mit Daniela Krüger, Dennis Pfisterer, Christian Haas und Denise Dudek im Rahmen des Projekts FleGSens konzipiert und evaluiert. Von den in FleGSens verwendeten Sicherheitsmechanismen wurden einige Mechanismen für diese Arbeit übernommen und an das DPS-Protokoll angepasst. Dies beinhaltet die Knotenausfallerkennung und den Schutz der Integrität und Authentizität des Nachrichteninhalts mittels AES-CBC-MAC.

2. Grundlagen

Dieses Kapitel stellt einige der zum Verständnis dieser Arbeit notwendigen Grundlagen vor. Zu Beginn wird hierbei in Abschnitt 2.1 auf das Themengebiet der drahtlosen Sensornetze eingegangen, wobei zunächst in Abschnitt 2.1.1 ein Überblick über einige der verfügbaren Hardwareplattformen gegeben wird, gefolgt von einer Übersicht über die verfügbaren Betriebssysteme in Abschnitt 2.1.2. Drahtlose Sensornetze sind für den Einsatz in einer Vielzahl von Anwendungsgebieten geeignet, was in Abschnitt 2.1.3 unter anderem am Beispiel der beiden vom Autor dieser Arbeit am Institut für Telematik bearbeiteten Projekte FleGSens und SmartAssist verdeutlicht wird. Aus den verwendeten Hard- und Softwareplattformen ergeben sich mehrere Eigenschaften von drahtlosen Sensornetzen, die in Abschnitt 2.1.4 zusammengefasst sind und sich unter dem Begriff Ressourcenbeschränkung zusammenfassen lassen. Um die aus dem jeweiligen Anwendungsgebiet resultierenden Anforderungen trotz dieser Ressourcenbeschränkung erfüllen zu können, sind die einzelnen Sensorknoten des Netzwerks auf eine Vielzahl von Kooperationsmechanismen angewiesen, die in Abschnitt 2.1.5 beschrieben werden.

Den zweiten Teil dieses Kapitels bildet eine Einführung in den Themenbereich Schichtenmodelle, die am Beispiel des ISO/OSI-Schichtenmodells und des TCP/IP-Referenzmodells in Abschnitt 2.2 vorgestellt werden. Neben der geschichtlichen Entwicklung sowie der Vorteile dieser Modelle wird hierbei auf die individuellen Schichten sowie deren Implementierung in der Form von Kommunikationsprotokollen eingegangen. Den Abschluss dieses Kapitels bildet eine Übersicht über die Verwendung von Schichtenmodellen im Umfeld der drahtlosen Sensornetze in Abschnitt 2.2.2.

Der letzte Teil dieses Kapitels beinhaltet einen Überblick über das Themengebiet der Remote-Procedure-Calls.

2.1. Drahtlose Sensornetze

Unter einem drahtlosen Sensornetz (engl.: Wireless Sensor Network, kurz: WSN) versteht man ein drahtloses Kommunikationsnetz, das aus Sensorknoten (kurz: Knoten) besteht, die jeweils mit einem oder mehreren Sensoren ausgestattet sind. Mittels dieser Sensoren können die Knoten physikalische Umweltparameter wie z.B. die Temperatur, Luftfeuchtigkeit oder Helligkeit messen um diese z.B. kontinuierlich oder beim Eintreten bestimmter Ereignisse an eine Basisstation weiterzuleiten.

Es existiert eine Vielzahl von unterschiedlichen Hard- und Softwareplattformen, die zum Betrieb eines WSNs eingesetzt werden können. Eine Übersicht findet sich in den beiden nachfolgenden Abschnitten 2.1.1 und 2.1.2, gefolgt von einer Zusammenfassung mehrerer typischer Anwendungsgebiete für WSNs. Basierend auf den Anforderungen der Anwendungsgebiete und der verwendeten Hard- und Software ergeben sich die in Abschnitt 2.1.4 beschriebenen allgemeinen Eigenschaften und Beschränkungen von WSNs. Viele Forschungsarbeiten haben sich bereits damit beschäftigt, wie diesen Beschränkungen durch die Kooperation benachbarter Sensorknoten zumindest teilweise entgegengewirkt werden kann. Aus diesem Grund beschäftigt sich Abschnitt 2.1.5 mit einem Überblick über existierende Lösungen und Ansätze zur Kooperation in Sensornetzen.

2.1.1. Hardwareplattformen

In diesem Abschnitt werden einige der bekanntesten und am weitesten verbreiteten Hardwareplattformen vorgestellt, die zum Betrieb eines WSNs verwendet werden können. Die zur Bildung des drahtlosen Sensornetzes verwendete Hardwareplattform besteht hierbei prinzipiell aus den folgenden fünf Komponenten:

Ein **Prozessor**, der für die Ausführung der Programmlogik zuständig ist und somit alle Abläufe im Netzwerk steuert. Als Prozessor kommen in Sensornetzen häufig Mikrocontroller (MCU, englisch: Microcontroller Unit) wie der MSP430 von Texas Instruments oder der Atmel ATmega128 zum Einsatz. Es werden häufig 16-Bit Mikrocontroller eingesetzt, aber auch 8- oder 32-Bit Mikrocontroller finden Verwendung. Die Taktrate der Mikrocontroller beträgt häufig nur wenige Megahertz (8-32 MHz), da für einen Sensorknoten ein niedriger Energieverbrauch wichtiger ist als eine hohe Taktrate.

Der **Speicher**, der zur Speicherung des Programmcodes notwendig ist. Hierfür stehen häufig zwei unterschiedliche Arten von Speicher zur Verfügung - nichtflüchtiger aber langsamer Flash-Speicher zur langfristigen Speicherung des Programmcodes und flüchtiger aber schnellerer RAM (Random Access Memory) als Arbeitsspeicher, der für die Ausführung des Programmcode nach dem Start des Sensorknoten benötigt wird.

Die **Sensoren**, die zur Beobachtung von unterschiedlichen Umgebungsdaten verwendet werden. Welche Sensoren eingesetzt werden, hängt vom jeweiligen Anwendungsgebiet ab (siehe Abschnitt 2.1.3). Typisch für Sensornetze sind jedoch vor allem Temperatur-, Luftfeuchtigkeits- und Helligkeitssensoren, da diese eine große Menge an Einsatzmöglichkeiten bieten.

Damit die Sensorknoten ein Netzwerk bilden können, besitzt jeder Knoten eine **Funkschnittstelle**, welche die drahtlose Kommunikation zwischen den Knoten ermöglicht. Hierbei hat sich im Laufe der Zeit vor allem IEEE 802.15.4 [10] als einer der De-facto-Standards erwiesen, der von einer großen Anzahl von Hardwareplattformen unterstützt wird. Neben IEEE 802.15.4 werden aber auch Bluetooth [11], WirelessHART [12] oder proprietäre Lösungen verwendet. Um größere Strecken zu überwinden oder um eine Anbindung an andere Computer-

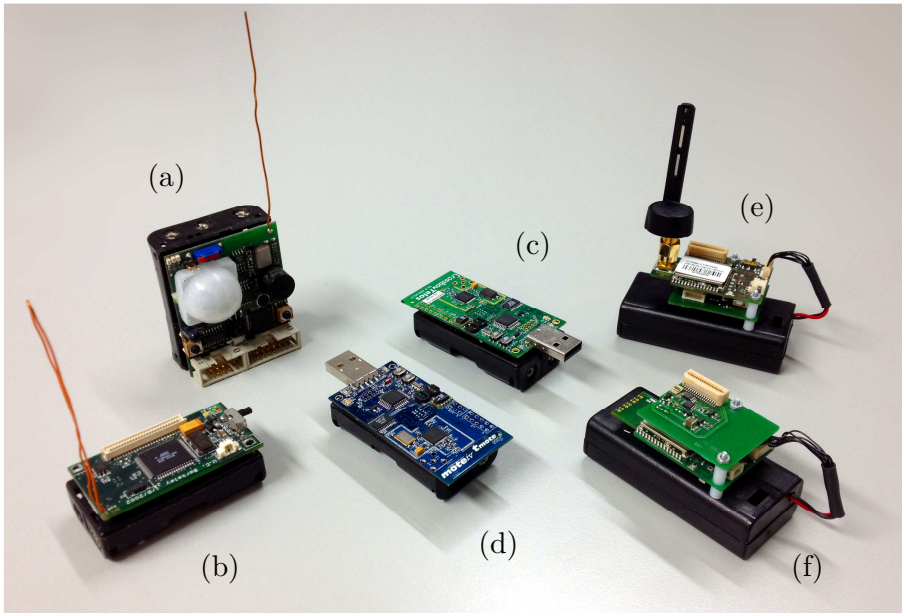


Abbildung 2.1.: Eine Auswahl unterschiedlicher Sensorknoten: (a) ScatterWeb ESB (b) UC Berkeley MICA2 (c) Crossbow TelosB (d) Moteiv Tmote Sky (e) iSense JN5139 mit SMA Antenne (f) iSense JN5148 mit PCB Antenne

netzwerke zu ermöglichen, können einzelne Knoten im Netzwerk (typischerweise die Basisstation) auch GSM/UMTS oder WLAN verwenden.

Als letzte Komponente besitzen Sensorknoten eine **Energieversorgung**, die häufig durch eine (wiederaufladbare) Batterie realisiert wird. Auf diese Weise können Sensorknoten genau dort eingesetzt werden, wo sie benötigt werden. Eine der am häufigsten eingesetzten Energiequellen sind hierbei zwei AA Batterien, da diese eine sehr hohe Verfügbarkeit aufweisen und niedrige Kosten verursachen. Sind größere Energiespeicher oder das Wiederaufladen der Batterie notwendig, werden häufig Lithium-Ionen-Akkumulatoren eingesetzt. Zum Wiederaufladen des Energiespeichers kann entweder eine Solarzelle oder eine andere Form des Energy-Harvesting verwendet werden.

Eine Auswahl an Beispielen für Hardwareplattformen, wie sie häufig in WSN eingesetzt werden, findet sich in Abbildung 2.1. Diese werden im Folgenden kurz vorgestellt. Weitere Übersichten über Hardwareplattformen für drahtlose Sensornetze finden sich in [7], [13] und [14].

ScatterWeb ESB

Die ScatterWeb ESB (Embedded Sensor Board [15]) Knoten wurden an der Freien Universität Berlin entwickelt. Der ESB verwendet den TI MSP430 F149, einen 16-Bit MCU, der mit 8 MHz getaktet ist und über 60 kB Flash und 2 kB RAM verfügt. Der ESB ist mit einem Temperatur-, Luftfeuchtigkeits-, Licht-

und Vibrationssensor sowie einen Bewegungsmelder und einem Mikrophon ausgestattet. Die Funkschnittstelle verwendet einen TR1001 Chip, der mit einer Frequenz von 868 MHz und einer Geschwindigkeit von 19,2 kbit/s sendet. Als Energieversorgung dienen 3 AA Batterien.

UC Berkeley MICA2

Die Sensorknoten vom Typ MICA2 [16] wurden an der University of California in Berkeley entwickelt und verwenden einen Atmel ATmega128L. Dieser 8-Bit Mikrocontroller ist mit 7,4 MHz getaktet und verfügt über 128 kB Flash und 4 kB RAM. Der MICA2 selbst besitzt keine eingebauten Sensoren, kann aber über Sensormodule erweitert werden, die z.B. mit einen Temperatur- und Luftdrucksensor, einem Beschleunigungssensor oder einem Mikrophon ausgestattet sind. Die Funkschnittstelle verwendet einen CC1000 Chip, der mit einer Frequenz von 868/916, 433 oder 315 MHz und einer Geschwindigkeit von 38,4 kbit/s sendet. Als Energieversorgung dienen 2 AA Batterien.

Crossbow TelosB und Moteiv Tmote Sky

Der TelosB [17] wurde von der Firma Crossbow (Heute: MEMSIC) in San Jose, Kalifornien entwickelt und ist eine Weiterentwicklung des TelosA. Der Tmote Sky hingegen wurde von Moteiv (Heute: Sentilla) in San Francisco, Kalifornien entwickelt und basiert auf dem TelosB. Beide Sensorknoten verwenden den 16-Bit Mikrocontroller TI MSP430 F1611, der mit 8 MHz getaktet ist und über 48 kB Flash und 10 kB RAM verfügt. Zusätzlich sind beide Knoten mit Sensoren für Temperatur, Licht und Luftfeuchtigkeit ausgestattet. Die Funkschnittstelle verwendet den CC2420 Chip, der nach dem 802.15.4-Standard mit einer Frequenz von 2,4 GHz und einer Geschwindigkeit von 250 kbit/s sendet. Als Energieversorgung dienen 2 AA Batterien.

iSense JN5139 und JN5148

Die Sensorknoten vom Typ iSense wurden von der Firma coalesenses, einer Ausgründung aus dem Institut für Telematik der Universität zu Lübeck, entwickelt. Sie verwenden den JN5139 [18] und JN5148 [19] der Firma Jennic (heute: NXP), einen 32-Bit Mikrocontroller, der mit 16 MHz getaktet ist. Der JN5139 besitzt 128 kB Flash und 96 kB RAM, während der neuere JN5148 über 512 kB Flash und 128 kB RAM verfügt. Im Gegensatz zu den oben vorgestellten Hardwareplattformen müssen diese Mikrocontroller nach dem Startvorgang ihr gesamtes Programm zur Ausführung zunächst vom Flash-Speicher in den RAM kopieren. Aus diesem Grund können auf der iSense-Plattform nur 96 bzw. 128 kB des Flash-Speichers für den Programmcode verwenden werden.

Die iSense-Sensorknoten besitzen keine eingebauten Sensoren, können aber über Module erweitert werden, die z.B. mit Sensoren zur Messung der Temperatur,

der Luftfeuchtigkeit, oder des Luftdrucks ausgestattet sind. Darüber hinaus können auch ein Beschleunigungssensor, ein Bewegungsmelder oder ein GPS Modul verwendet werden. Die Funkschnittstelle wurde in den Mikrocontroller integriert und arbeitet nach dem 802.15.4-Standard mit einer Frequenz von 2,4 GHz und einer Geschwindigkeit von 250 kbit/s. Die Funkschnittstelle kann mit verschiedenen Antennen ausgestattet werden - in Abbildung 2.1 sind die SMA und PCB Antenne dargestellt, darüber hinaus existiert noch eine Keramikantenne. Als Energieversorgung können neben 2 AA Batterien auch verschiedene Lithium-Ionen-Akkus verwendet werden, die mit einer Solarzelle kombiniert werden können, die das Wiederaufladen des Akkus ermöglicht.

Weitere Hardwareplattformen

Neben den oben vorgestellten Plattformen existieren noch weitere Sensorknoten, die für unterschiedliche Anwendungsgebiete entwickelt wurden. Als Mikrocontroller findet hierbei häufig ebenfalls ein MSP430 (Eyes [20], SenseNode [21], EPIC Mote [22], Tinynode [23]) oder ATmega128 (AVR Raven [24], BTnode [25], Iris Mote [26], Waspmote [27], Arduino [28]) Verwendung. Neben diesen beiden weit verbreiteten Mikrocontrollern finden sich auch Sensorknoten auf Basis eines ARM Mikrocontrollers, wie z.B. der SunSpot [29], Egs [30] oder Preon32 [31].

2.1.2. Softwareplattformen

Neben den im vorangegangenen Abschnitt vorgestellten Hardwareplattformen spielt die verwendete Software eine wichtige Rolle bei der Realisierung eines Sensornetzes. Hierbei haben sich mehrere große Betriebssysteme entwickelt, die in diesem Abschnitt vorgestellt werden. Neben dem Betriebssystem werden zusätzlich die verfügbaren IPv6-Implementierungen der jeweiligen Plattform vorgestellt und es wird auf die Kompatibilität zu den oben genannten Hardwareplattformen eingegangen. Weitere Informationen über Betriebssysteme für drahtlose Sensornetze finden sich in [32], [33] und [34]. Eine Übersicht über die verfügbaren IPv6-Implementierungen findet sich unter anderem in [35], [36] und [37].

TinyOS

Das Betriebssystem TinyOS[1] entstand ab dem Jahr 2000 unter der Leitung von David Culler [38, 39] als eine Zusammenarbeit zwischen der University of California in Berkeley, Intel Research (ebenfalls Berkeley) und der Firma Crossbow aus San Jose in Kalifornien. TinyOS ist quelloffen und steht unter der BSD Lizenz.

Anwendungen für TinyOS werden in nesC geschrieben, einer komponentenbasierten und ereignisgesteuerten Programmiersprache, die eine Anpassung

[1] http://www.tinyos.net/

der Programmiersprache C für Sensorknoten darstellt. Ein Programm basiert aus mehreren Komponenten, die als endlicher Automat modelliert sind. Zustandsübergänge dieses Automaten werden durch Ereignisse, wie z.B. das Eintreffen einer Nachricht über die Funkschnittstelle, das Messen eines Sensorwertes oder das zeitlich gesteuerte Ausführen eines Tasks ausgelöst. Auf diese Weise versucht TinyOS, das Programmiermodell den Anforderungen von drahtlosen Sensornetzen anzupassen: Eintretende Ereignisse werden sofort und schnell behandelt, um die Energiesparfunktionen der Hardware über möglichst lange Zeiträume nutzen zu können.

TinyOS unterstützt Mikrocontroller vom Typ TI MSP430 sowie unterschiedliche Atmel Chips wie den ATmega128, ATmega128L und ATmega1281[2]. Von den in Abschnitt 2.1.1 aufgezählten Hardwareplattformen werden der TelosB, Mica2 und Tmote Sky unterstützt. Es existieren darüber hinaus Portierungen für MicaZ, IRIS, Shimmer, Epic, Mulle, Tinynode, Span und iMote2.

Die Entwicklung des IPv6-Stack für das TinyOS-Betriebssystems wurde ursprünglich unter dem Namen b6loWPAN im Jahr 2008 gestartet und später in BLIP (Berkeley Low-Power IP Stack) umbenannt [36]. Die aktuelle Version ist BLIP 2.0[3], die einen vollständigen IPv6-Stack für TinyOS darstellt und IPv6 mit Neighbor-Discovery, sowie die 6LoWPAN-Adaptionsschicht unterstützt. Als Routing-Protokoll verwendet BLIP TinyRPL [40], eine Implementierung des RPL-Routing-Protokoll für TinyOS. Darüber hinaus existieren eine Vielzahl von Transport- und Anwendungsprotokollen, wie z.B. UDP, TCP und CoAP [41].

Contiki

Die Entwicklung von Contiki[4] begann im Jahr 2002 unter der Leitung von Adam Dunkels am SICS (Swedish Institute of Computer Science) in Stockholm [42, 43]. Bis heute haben mehrere internationale Partner an Contiki mitgearbeitet, unter anderem die ETH Zürich, die RWTH Aachen, die University of Oxford und unterschiedliche Firmen wie z.B. Atmel, Cisco, SAP und Sensinode. Contiki ist quelloffen und steht unter der BSD Lizenz.

Die Programmierung für Contiki erfolgt in der Programmiersprache C und verwendet Protothreads [44]. Protothreads sind eine leichtgewichtige Form von Threads: Sie erlauben Nebenläufigkeit und Multithreading auch auf stark ressourcenbeschränkten Geräten, indem der Speicherverbrauch durch den Verzicht auf lokale Variablen und einen eigenen Stack für jeden Thread reduziert wird. Contiki unterstützt Mikrocontroller vom Typ TI MSP430 sowie die Atmel Chips ATmega128 und AVR. Von den in Abschnitt 2.1.1 aufgezählten Hardwareplattformen werden der TelosB [45] und Tmote Sky [46] unterstützt. Darüber hinaus existiert eine Portierung für die JN5139- und JN5148-Plattform[5].

[2] http://tinyos.stanford.edu/tinyos-wiki/index.php/Platform_Hardware
[3] http://tinyos.stanford.edu/tinyos-wiki/index.php/BLIP_2.0
[4] http://www.contiki-os.org/
[5] Jennisense, siehe: https://github.com/teco-kit/Jennisense/

Der IP-Stack des Contiki-Betriebssystems heißt uIP [36] und unterstützt neben IPv4 und IPv6 auch ICMP, UDP und TCP. Die 6LoWPAN-Schicht wurde am SICS entwickelt und als SICSlowpan [47] in das Betriebssystem integriert. Für SICSlowpan/Contiki existieren drüber hinaus Implementierungen des Routing-Protokolls RPL [40] und des Anwendungsprotokolls CoAP [48].

iSense

Die Entwicklung des iSense-Betriebssystems[6] begann ab dem Jahr 2007 am Institut für Telematik der Universität zu Lübeck. Mittlerweile wird das Betriebssystem zusammen mit der Sensorknotenhardware von der Firma coalesenses vertrieben. iSense ist nicht quelloffen und nur in einer vorkompilierten Version frei verfügbar. Zugriff auf den Quelltext ist über eine Geheimhaltungserklärung möglich.

Die Programmierung des Betriebssystems und der Anwendungen geschieht objektorientiert mittels C++. Das Betriebssystem unterstützt dynamische Speicherverwaltung, Hardwareabstraktion und Treiber für verschiedene Sensoren sowie eine Vielzahl von Sensornetz-spezifischen Protokollen (Lokalisierung, Zeitsynchronisation, ...). iSense unterstützt hauptsächlich die Mikrocontroller der Firma Jennic, wie den JN5139 und JN5148. Es existieren jedoch Portierungen von iSense für den Pacemate (LPC2136 ARM7) und den TelosB[7]. Offiziell unterstützt und weiterentwickelt wird zum aktuellen Zeitpunkt lediglich die Version für den JN5148.

Mit der Entwicklung des iSense-IPv4/6-Stack wurde im Rahmen einer Diplomarbeit [49] am Institut für Telematik begonnen, die vom Autor dieser Arbeit in Zusammenarbeit mit Carsten Buschmann von der Firma coalesenses betreut wurde. Der iSense-IPv6-Stack unterstützt die 6LoWPAN-Schicht und die beiden Routing-Protokolle DYMO [50] und DYMO-low [51]. Darüber hinaus stehen UDP, TCP als Transportprotokolle sowie die Anwendungsprotokolle HTTP und CoAP zur Verfügung.

Andere Betriebssysteme

Zusätzlich zu den oben vorgestellten Systemen existieren noch weitere Betriebssysteme für drahtlose Sensornetze. Dazu zählen unter anderem Mantis[8] von der University of Colorado at Boulder (ATmega128, MSP430), FreeRTOS[9] von der Firma Real Time Engineers Ltd. aus London (MSP430, ARM, AVR), LiteOS[10] von der University of Illinois in Urbana and Champaign (ATmega128, AVR) oder Nano-RK[11] von der Carnegie Mellon University (ATmega128).

[6] http://www.coalesenses.com/
[7] http://www.coalesenses.com/index.php/webcompile/
[8] http://mantisos.org
[9] http://www.freertos.org/
[10] http://www.liteos.net/
[11] http://www.nanork.org

2.1.3. Anwendungsgebiete

Drahtlose Sensornetze eignen sich für eine Vielzahl von unterschiedlichen Anwendungsgebieten, bei denen die flächendeckende Überwachung von physikalischen Phänomenen oder Objekten notwendig ist. Durch die drahtlose Kommunikation und unabhängige Stromversorgung sind die Sensorknoten an keine spezielle Infrastruktur gebunden und können flexibel genau dort eingesetzt werden, wo sie benötigt werden. Dies ermöglicht den Einsatz eines Sensornetzes in der freien Natur oder die nachträgliche Installation innerhalb eines Gebäudes, ohne dass bauliche Veränderungen notwendig sind.

Die Art der Überwachung durch drahtlosen Sensornetzen kann hierbei in zwei Bereiche unterteilt werden: Langzeitbeobachtung und Ereigniserkennung. Bei der Langzeitbeobachtung sammeln die Sensorknoten kontinuierlich Daten und senden diese zu einer Basisstation, bei der die Daten gesammelt und ausgewertet werden. Dieser Sendevorgang kann z.B. automatisch in regelmäßigen Abständen oder nur auf Anfrage ausgelöst werden. Ein Beispiel hierfür sind Wetterstationen, welche die Temperatur, Niederschlagsmenge, Sonnenscheindauer sowie Windrichtung und -geschwindigkeit in regelmäßigen Abständen messen und diese zur Anzeige und Auswertung an eine zentrale Sammelstelle weiterleiten. Andere Anwendungsgebiete sind z.B. die Gebäudeautomation, bei der Sensorknoten zur Steuerung der Klimaanlage und Beleuchtung verwendet werden können oder die kontinuierliche Überwachung der Position von Wild- oder Nutztieren mittels GPS.

Bei der Ereigniserkennung hingegen beobachten die Sensorknoten ein physikalisches Phänomen und melden nur das Eintreten eines bestimmten Ereignisses an eine Basisstation, wie z.B. das über- oder unterschreiten bestimmter Sollwerte. Ein Beispiel hierfür ist die Überwachung der Luftqualität in Gefahrenbereichen. Steigt hierbei die Konzentration eines schädlichen Gases über eine bestimmte Grenze, so wird ein Alarm ausgelöst und an die Basisstation weitergeleitet. Hierfür findet eine Vorverarbeitung der Messergebnisse im Sensornetz statt, die neben der Filterung auch das Zusammenfassen (Aggregation) der Werte beinhalten kann. Eine mögliche Anwendung in der Logistik besteht darin, einzelne Sendungen mit Sensorknoten auszustatten, die an den Umladestationen zusätzlich zur Identifikation und Aufenthaltsortbestimmung auch zur Überwachung des ordnungsgemäßen Transports (z.B. Erschütterungen, Kühlkette) verwendet werden können.

Um einen besseren Überblick über die Anwendungsgebiete von drahtlosen Sensornetzen zu bieten, werden im Folgenden sowohl die Langzeitüberwachung als auch die Ereigniserkennung anhand von zwei Beispielen näher erläutert. Die beiden Beispiele stellen hierbei Projekte dar, an denen der Autor dieser Arbeit während seiner Zeit am Institut für Telematik aktiv mitgearbeitet hat. Zunächst wird die Langzeitbeobachtung am Beispiel des Projekts SmartAssist [52] beschrieben, anschließend folgt die Ereigniserkennung am Beispiel des Projekts FleGSens [53].

Langzeitbeobachtung in SmartAssist

Das Projekt SmartAssist wurde in der Zeit von August 2009 bis Dezember 2012 vom Bundesministerium für Bildung und Forschung gefördert (Förderkennzeichen 16KT0942). Ziel des Projekts war die Schaffung einer Plattform zur Unterstützung des altersgerechten autonomen Lebens: Anstelle einer frühzeitigen Einweisung in ein Heim, in dem eine dauerhafte Überwachung geleistet wird, soll eine bedarfsorientierte Betreuung alter Menschen in ihrer eigenen Wohnung erfolgen.

Der in SmartAssist verfolgte Ansatz sieht hierbei den Einsatz eines drahtlosen Sensornetzes im häuslichen Umfeld vor, das als Basis für die Erkennung von gesundheitsrelevanten Zuständen dient. Abbildung 2.2 zeigt eine Übersicht der im Projekt verwendeten Sensoren, die z.B. den Wasserverbrauch im Badezimmer und in der Küche sowie die Temperatur und Luftfeuchtigkeit in den einzelnen Zimmern der Wohnung messen können. Darüber hinaus wird kontinuierlich der Aufenthaltsort der Person in der Wohnung über Bewegungsmelder und Türsensoren festgestellt.

Die Sensorknoten sammeln an sieben Tagen in der Woche und 24 Stunden am Tag ihre Messwerte und leiten diese im Sensornetz an eine Basisstation weiter. Von dort werden die Daten über das Internet an einen Server zur Speicherung und Auswertung übertragen. Im Rahmen der Auswertung werden die gesammelten Messwerte zur Modellierung des Tagesablaufs der Benutzer verwendet. Das Ziel dieser Auswertung ist es, schleichende Veränderungen im Tagesablauf festzustellen, die als Indikatoren für bestimmte Alterserkrankungen dienen können. Diese Indikatoren können dann vom Hausarzt bei der nächsten Untersuchung bei der Diagnose berücksichtigt werden.

Die Anforderungen an das drahtlose Sensornetz sind hierbei eine zuverlässige Erfassung und Weiterleitung aller Messergebnisse, eine besonders lange Batterielaufzeit, die flexible und nicht störende Anbringung der Sensoren sowie der Schutz der Daten vor unberechtigtem Zugriff oder Veränderung.

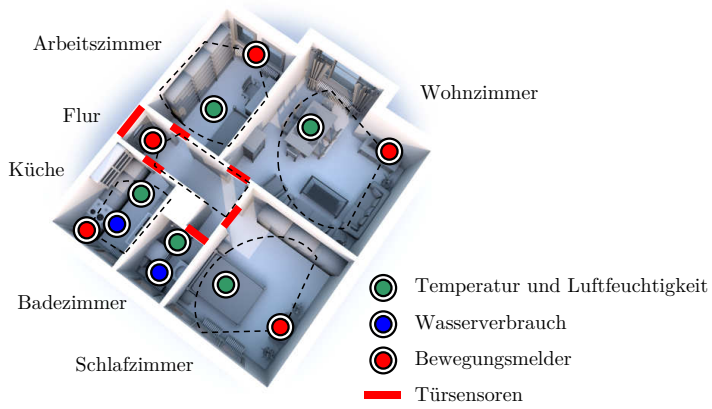


Abbildung 2.2.: Übersicht über die in SmartAssist verwendeten Sensoren

Ereigniserkennung in FleGSens

Das Projekt FleGSens wurde in der Zeit von Januar 2008 bis Juli 2009 vom Bundesamt für Sicherheit in der Informationstechnik gefördert. Das Projekt beschäftigte sich mit der Konzeption, Entwicklung und Erprobung eines drahtlosen Sensornetzes zur Grenzüberwachung. Das Sensornetz hat hierbei die Aufgabe, das unberechtigte Überqueren eines zuvor festgelegten Grenzbereiches mit hinreichender Genauigkeit zu detektieren, lokalisieren und analysieren. In FleGSens wurden insbesondere Mechanismen zur Gewährleistung der Ausfall- und Informationssicherheit implementiert. Dies schließt den Schutz vor möglichen Angreifern ein, die z.B. durch die aktive Manipulation der drahtlosen Netzkommunikation versuchen, das System außer Betrieb zu setzen. Intention des Projekts war es, die Realisierbarkeit solcher Sicherheitsmechanismen auf ressourcenbeschränkter Hardware, wie sie in Sensornetzen zum Einsatz kommt, zu demonstrieren.

Im Rahmen des Projektes wurden alle hierfür notwendigen Protokolle zunächst konzipiert und entwickelt, in Simulationen mit bis zu 2000 Sensorknoten getestet und abschließend mehrmals in einem Demonstrator unter realen Bedingungen getestet. Eine Version dieses Demonstrators ist in Abbildung 2.3 dargestellt und umfasst ca. 200 Sensorknoten in einem Park in der Nähe der Universität zu Lübeck. Die Sensorknoten erhalten ihre Energie aus einem Lithium-Ionen-Akku, der mittels einer Solarzelle geladen werden kann, und sind mit einem Passiv-Infrarot Bewegungsmelder ausgestattet. Mittels dieses Bewegungsmelders überwacht jeder Sensorknoten einen etwa 10 m langen und 16 m breiten kegelförmigen Bereich. Benachbarte Sensorknoten tauschen Informationen über Sensorereignisse aus und erzeugen aus mehreren dieser Ereignisse eine Alarmnachricht, die dann an eine Basisstation gemeldet wird. Anstatt alle erkannten Ereignisse an einer Basisstation zu sammeln und dort zu verarbeiten, aggregieren die benachbarten Sensorknoten räumlich und zeitlich benachbarte Ereignisse zu Alarmmeldungen.

FleGSens erlaubt es, einen Eindringling innerhalb von 5 s mit einer Genauigkeit von 10 m zu detektieren und bei der Basisstation zu melden. Durch den Einsatz von Energiespartechniken kann FleGSens auch ohne Solarzellen über eine Dauer von sieben Tagen eingesetzt werden.

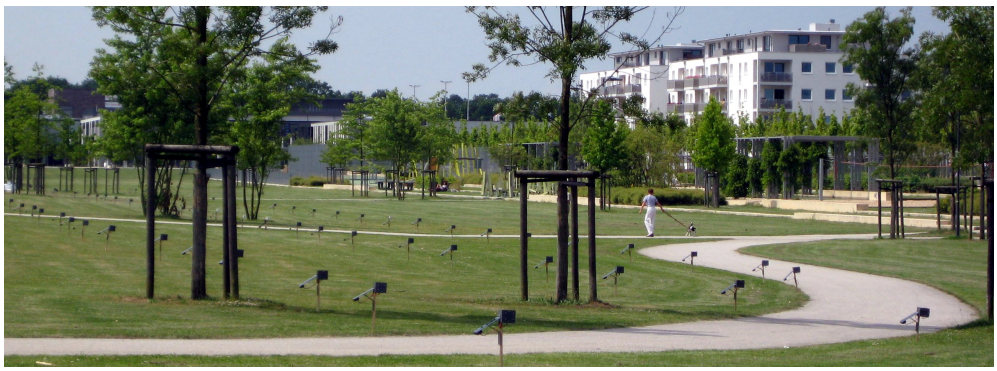


Abbildung 2.3.: FleGSens - ein drahtloses Sensornetz zur Grenzüberwachung

2.1.4. Eigenschaften

Basierend auf den Eigenschaften der in Abschnitt 2.1.1 beschriebenen Hard- und Softwareplattformen in Zusammenhang mit den Anforderungen der Anwendungsgebiete aus Abschnitt 2.1.3 ergeben sich mehrere grundlegende Eigenschaften von drahtlosen Sensornetzen, die in diesem Abschnitt zusammengefasst sind. Hierbei wird unter anderem auf die Ressourcenbeschränkungen von Sensornetzen, zunächst in Bezug auf Speicherplatz, Rechenleistung und Energieversorgung sowie anschließend auf die drahtlose Kommunikation eingegangen.

Ressourcenbeschränkungen

Bei der Entwicklung einer Anwendung für Sensornetze muss darauf geachtet werden, dass die Ressourcen, die in einem Sensornetz zur Verfügung stehen, stark beschränkt sind. Hierzu zählen in erster Linie die Rechenleistung und der Speicherbedarf: Typisch für die 8 oder 16-Bit Mikrocontroller sind Taktraten von nur wenigen Megahertz und Speicherausstattungen von nur 10 kB Arbeitsspeicher und 128 kB Programmspeicher. Aus diesem Grund eignen sich Sensornetze nicht für Mustererkennungsalgorithmen, die z.B. Objekte in Videobildern erkennen sollen, oder sie sind auf Zusatzhardware angewiesen.

Auch die Energieversorgung eines Sensorknoten ist beschränkt und bedingt somit den Einsatz von Energiesparmechanismen, die z.B. die Funkschnittstelle im Rahmen eines Duty-Cycle-Protokolls nur bei Bedarf aktivieren. So wird z.B. dem Mica2 Sensorknoten in seinem Datenblatt[12] eine Laufzeit von mehr als einem Jahr mit nur zwei AA Batterien attestiert.

Drahtlose Kommunikation

Sensorknoten kommunizieren untereinander über eine Funkschnittstelle. Diese Funkschnittstelle verfügt nur über eine begrenzte Reichweite zwischen 10 und 100 m, so dass nicht alle Sensorknoten in einem Netzwerk direkt miteinander kommunizieren können. Stattdessen ist es häufig notwendig, dass Nachrichten von einem Sensorknoten über eine oder mehrere Zwischenstationen bis zum Ziel weitergeleitet werden müssen. Die drahtlose Kommunikation kann hierbei durch eine Vielzahl von physikalischen Effekten negativ beeinträchtigt werden, wie z.B. Interferenz, Dämpfung, Reflexionen oder Funkschatten. Durch die zeitliche Veränderung dieser Effekte kann die Bitfehler- und Paketverlustrate stark schwanken - häufig kommt es zum unvorhergesehenen Abbruch einer vorher stabilen Verbindung.

Diese Effekte in Kombination mit den oben erwähnten Energiesparmechanismen können die ohnehin schon niedrige Bandbreite zwischen 20 und 250 kbit/s noch weiter reduzieren und auch die Latenz zwischen Sender und Empfänger erhöhen.

[12] http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf

Selbstorganisation und Skalierbarkeit

Um ein möglichst großes Gebiet mit möglichst hoher räumlicher Auflösung mit Hilfe eines drahtlosen Sensornetzes zu überwachen, wird eine große Anzahl von Sensorknoten benötigt. Die Konfiguration und Verwaltung von mehreren hundert oder sogar tausend Sensorknoten stellt hierbei eine schwierige Aufgabe dar.

Aus diesem Grund müssen Sensornetze mit Algorithmen und Protokollen versehen werden, die einerseits skalierbar sind, d.h. auch in großen Netzen noch funktionieren, und andererseits Mechanismen zur Selbstorganisation beinhalten. Diese häufig als Self-X bezeichneten Eigenschaften schließen einerseits die selbständige Organisation und Konfiguration (Self-Organization und Self-Configuration) ein, welche die Einrichtung und den Betrieb des Sensornetzes vereinfachen, umfassen jedoch auch Eigenschaften wie Self-Optimization und Self-Healing. Die beiden letzteren Eigenschaften ermöglichen, dass ein Routing-Protokoll im Sensornetz ohne den manuellen Eingriff eines Menschen den besten Weg von A nach B wählt und auch auf Störungen in der drahtlosen Kommunikation reagiert.

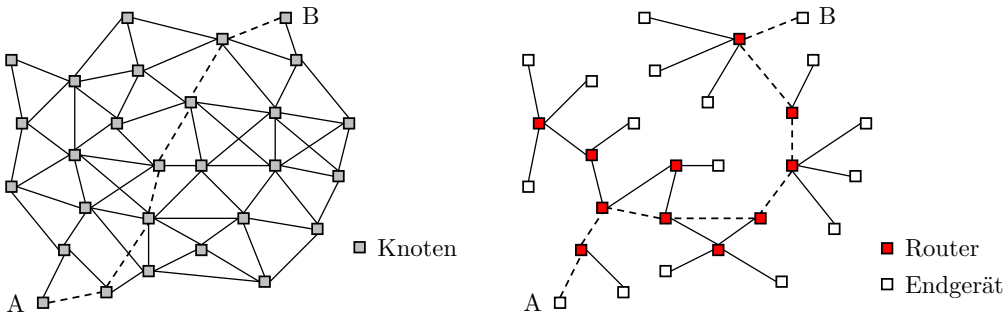
Diese Eigenschaften sorgen dafür, dass der Ausfall, das Austauschen oder das Hinzufügen einzelner Sensorknoten im Netzwerk keine langfristigen negativen Auswirkungen auf die Leistung des Sensornetzes hat.

2.1.5. Kooperation und Aufgabenteilung in Sensornetzen

Wie im vorangegangenen Abschnitt beschrieben wurde, müssen bereits bei der Konzeption von Algorithmen und Protokollen für Sensornetze alle Ressourcenbeschränkungen bedacht werden. Dies hat zu der Entwicklung einer Vielzahl von Mechanismen geführt, welche die Kooperation benachbarter Sensorknoten ausnutzen, um diesen Ressourcenbeschränkungen entgegen zu wirken. Kooperation in Sensornetzen ist auf unterschiedliche Arten möglich, von deinen einige im Folgenden vorgestellt werden sollen. Das Grundprinzip ist hierbei, ein Gesamtproblem in mehrere Teilaufgaben zu unterteilen. Anstatt nun jeden Sensorknoten alle Teilaufgaben im Netzwerk erfüllen zu lassen, werden die Aufgaben auf unterschiedliche Klassen von Sensorknoten verteilt.

Routing

Sensorknoten kooperieren bei der Weiterleitung von Nachrichten, indem sie eintreffende Nachrichten, die für ein anderes Ziel bestimmt sind, entlang eines kürzesten Pfades weiterleiten. Dieser kürzeste Pfad wird in der Regel dynamisch zur Laufzeit von einem Routing-Protokoll ermittelt. Hierfür sammelt das Routing-Protokoll eine Liste der Verbindungen zwischen allen Sensorknoten im Netzwerk, ermittelt die Kosten der einzelnen Verbindungen und berechnet den kürzesten Pfad. Ein Beispiel für dieses Vorgehen ist in Abbildung 2.4 (a) dargestellt:



(a) (b)

Abbildung 2.4.: Kooperation beim Routing: Alle Knoten haben dieselben Aufgaben und bilden ein vermaschtes Netz (a) im Vergleich zur Aufteilung in Router und Endgeräte (b)

Dargestellt sind die Verbindungen zwischen allen Knoten im Netzwerk und der kürzeste Pfad zwischen Knoten A (links unten) und Knoten B (rechts oben). Aufgrund der großen Anzahl an Verbindungen sind in diesem Netzwerk mehrere kürzeste Pfade zwischen diesen beiden Knoten möglich, die alle vom Routing-Protokoll berücksichtigt werden müssen. Darüber hinaus wird das Routing-Protokoll auf allen im Netzwerk vorhandenen Knoten ausgeführt, wodurch alle Sensorknoten dieselben Aufgaben erfüllen.

Eine Form der Aufgabenteilung besteht darin, nicht alle Sensorknoten am Routing teilnehmen zu lassen und die Menge der Knoten stattdessen in zwei Teilmengen aufzuteilen: Router und Endgeräte. Dies ist in Abbildung 2.4 (b) dargestellt: Router (rot) nehmen am Routing-Protokoll teil und können eintreffende Nachrichten an andere Knoten weiterleiten. Die Endgeräte (weiß) hingegen können nur mit genau einem in der Nähe befindlichen Router kommunizieren und nehmen nicht am Routing-Protokoll teil. Der kürzeste Pfad zwischen Knoten A und Knoten B verläuft folglich ausschließlich zwischen den Routern im Netzwerk und beinhaltet keine weiteren Endgeräte, außer Knoten A und B.

Dieser Ansatz hat eine weite Verbreitung in Netzwerken gefunden, wobei sich lediglich die Namen der beiden Knotenarten unterscheiden: Bereits im 802.15.4-Standard [10] werden Full-Function-Devices und Reduced-Function-Devices unterschieden. Der ZigBee-Standard [54] unterscheidet ZigBee-Router und ZigBee-End-Devices und auch RFC 2460 [8] zu IPv6 unterscheidet Router und Hosts.

Die effiziente Platzierung der Router im Netzwerk wurde in der Literatur bereits intensiv unter dem Begriff Relay-Node-Placement (RNP) untersucht [55] [56]. Die in diesem Abschnitt vorgestellte Vorgehensweise wird in diesem Kontext als Two-Tiered RNP (2T-RNP [57]) bezeichnet. Eine genauere Untersuchung des 2T-RNP findet im Rahmen der Evaluation in Abschnitt 6.1.1 statt.

Clustering

Ein weiterer Mechanismus, der zur Organisation eines Sensornetzes und zur Verbesserung seiner Lebensdauer genutzt werden kann, wird als Clustering bezeichnet (siehe [58], [59]). Hierbei werden die Sensorknoten in mehrere disjunkte Gruppen aufgeteilt, die als Cluster bezeichnet werden und aus einer Menge von Sensorknoten bestehen, die häufig eine räumliche Nachbarschaft zueinander aufweisen. Zusätzlich befindet sich unter den Sensorknoten eines Clusters ein sogenannter Cluster-Head, der für die Organisation des Clusters zuständig ist und weitere Aufgaben übernehmen kann. Zu diesen Aufgaben kann die Aggregation der gesammelten Sensorwerte des Clusters sowie die Weiterleitung der aggregierten Daten an eine Basisstation gehören. Auf diese Weise wird der Kommunikationsaufwand im Netzwerk reduziert, da nicht alle Sensorknoten direkt mit der Basisstation kommunizieren, sondern lediglich mit ihrem Cluster-Head. Der Cluster-Head filtert fehlerhafte oder redundante Daten heraus und sendet nur die aggregierten Daten an die Basisstation. Auf diese Weise werden weniger Nachrichten versendet, was zu einer Reduktion des Energieverbrauchs und somit zu einer Verlängerung der Lebensdauer des Netzwerks führt. Clustering-Algorithmen unterscheiden sich in der Größe der gewählten Cluster, der verwendeten Strategie zur Wahl des Cluster-Heads sowie die Anpassung der Cluster-Größe und des Cluster-Heads zur Laufzeit, z.B. in Abhängigkeit von der Entfernung zur Senke.

Ein Beispiel für ein solches Clustering-Verfahren ist LEACH [60]. LEACH fasst räumlich benachbarte Sensorknoten zu Clustern zusammen, in denen die gesammelten Sensorwerte aggregiert werden, und lässt zusätzlich die Rolle des Cluster-Heads in jedem Cluster zufällig rotieren, um den Energieverbrauch in den Clustern gleichmäßiger unter den Knoten zu verteilen und somit die Lebensdauer zu erhöhen. Es existieren mehrere Erweiterungen für LEACH, z.B. TL-LEACH [61], bei dem die Cluster in mehrere Sub-Cluster aufgeteilt werden, die jeweils einen Sub-Cluster-Head besitzen.

Weitere Ansätze

Kooperation zwischen Sensorknoten wird noch in vielen weiteren Bereichen eingesetzt. So kombiniert das Medienzugriffsverfahren aus [62] den verteilten Ansatz von CSMA/CA mit einem clusterbasierten RTS/CTS-Mechanismus. Um den Energieverbrauch beim Zugriff auf die gesammelten Sensorwerte zu minimieren, schlagen die Autoren aus [63] ein kooperatives Cache-Protokoll vor, bei dem die Sensorwerte im Netzwerk von sogenannten Mediatoren zwischengespeichert werden. Kooperation kann auch in Abhängigkeit von der Hardwareausstattung verwendet werden: Dieser Ansatz wird häufig bei Lokalisierungsalgorithmen [64] verwendet, bei denen nur eine Untermenge der Sensorknoten mit einem Modul zur Positionsbestimmung ausgestattet sind, während die anderen Knoten ihre Position durch Triangulation zu diesen Knoten schätzen.

2.1.6. Das WISEBED-Testbed

Die Entwicklung und der Test von Anwendungen für drahtlose Sensornetze unter realistischen Bedingungen werden durch mehrere Faktoren erschwert. Zum einen kann die Funktionsweise und die Skalierbarkeit vieler Algorithmen nur in Sensornetzen mit einer großen Anzahl von Sensorknoten gezeigt werden. Hieraus ergeben sich hohe Anschaffungs- und Wartungskosten für den Betreiber eines solchen Sensornetzes. Zum anderen sind die Programmierung und das Debugging der Anwendungen in einem Sensornetz mit Schwierigkeiten verbunden. Jeder einzelne Sensorknoten muss zunächst mit der gewünschten Anwendung programmiert werden. Dieser Arbeitsschritt muss zudem im Laufe der Entwicklung viele Male wiederholt werden, wofür häufig eine physikalische Verbindung (z.B. ein USB-Kabel) zu jedem einzelnen Sensorknoten notwendig ist. Es existieren auch drahtlose Arten der Programmierung von Sensorknoten - diese funktionieren in der Regel jedoch nur dann, wenn das Programm auf den Sensorknoten fehlerfrei läuft und nicht abstürzt, was während der Testphase einer Anwendung nicht zuverlässig gewährleistet werden kann. Darüber hinaus müssen häufig Statusinformationen über die Sensorknoten im Netz zur Kontrolle der Funktionalität gesammelt werden. Werden diese Daten drahtlos übertragen, so werden die zu testenden Protokolle durch den zusätzlichen Übertragungs-, Verarbeitungs- und Speicheroverhead negativ beeinflusst. Soll z.B. ein Routing-Protokoll getestet werden, so muss zusätzlich ein garantiert funktionsfähiges Routing-Protokoll zur Sammlung der Kontrollinformationen verwendet werden.

Aus diesen Gründen wurden in den letzten Jahren unterschiedliche Testbeds für drahtlose Sensornetze entwickelt. Unter einem Testbed versteht man eine Ansammlung von Sensorknoten, die über eine Schnittstelle auf einfache Art und Weise programmiert und getestet werden können. Diese Schnittstelle kann entweder über einen lokalen Rechner oder z.B. über das Internet verwendet werden. Beispiele für solche Testbeds sind unter anderem das MoteLab [65] von der Universität Harvard, TWIST [66] von der TU Berlin oder FlockLab [67] von der ETH Zürich. Eine Liste vieler verfügbarer Testbeds befindet sich unter [68].

Das WISEBED-Testbed [69] wurde im Rahmen des EU FP7 Projekts WISEBED[13] in der Zeit von Juni 2008 bis Mai 2011 von insgesamt neun europäischen Partnern entwickelt. Unter den Partnern befanden sich neben dem Institut für Telematik der Universität zu Lübeck unter anderem die ALG Gruppe des IBR der Technischen Universität Braunschweig sowie die Research Unit 1 des Computer Technology Institute aus Patras in Griechenland.

Das WISEBED-Testbed am Institut für Telematik besteht aus über 100 Sensorknoten, die über eine Webservice-Schnittstelle an das Internet angebunden sind und die kostenlos von Forschern auf der ganzen Welt für ihre Experimente genutzt werden können. Diese Schnittstelle erlaubt die Programmierung der Sensorknoten sowie das Senden und Empfangen von Kontrollnachrichten an die/von den Sensorknoten. Die Programmierung und Kommunikation mit den Sensorknoten erfolgt hierbei drahtgebunden über eine USB-Verbindung, so

[13] http://www.wisebed.eu

dass die Schnittstelle unabhängig von den zu testenden Protokollen arbeitet. Das WISEBED-Testbed wird in der vorliegenden Arbeit für die Multi-Hop-Experimente zur Evaluation des DPS-Protokolls in Kapitel 7 verwendet.

2.2. Schichtenmodelle

Als Schichtenmodell wird die Einteilung einer Softwarearchitektur in voneinander getrennte Schichten bezeichnet. Hierfür wird zunächst die Gesamtfunktionalität der Softwarearchitektur in seine Bestandteile zerlegt, die anschließend gruppiert und den einzelnen Schichten zugeordnet werden. Die Schichtenmodelle, die im Rahmen dieser Arbeit verwendet werden sollen, werden für die Kommunikation zwischen Computersystemen eingesetzt. Hierbei werden die einzelnen Kommunikationsprimitiven in aufeinander aufbauenden Schichten organisiert, um so z.B. von der Kanalkodierung über den Medienzugriff bis zur Ende-zu-Ende Flusskontrolle und Reihenfolgeerhaltung abstrahieren zu können. Dieser Abschnitt beschreibt zunächst die geschichtliche Entwicklung und stellt die beiden wichtigsten Vertreter, das TCP/IP-Referenzmodell und das ISO/OSI-Schichtenmodell vor. Anschließend werden die Vor- und Nachteile von Schichtenmodellen erläutert und in Abschnitt 2.2.2 auf die im Umfeld der drahtlosen Sensornetze eingesetzten Schichtenmodelle eingegangen.

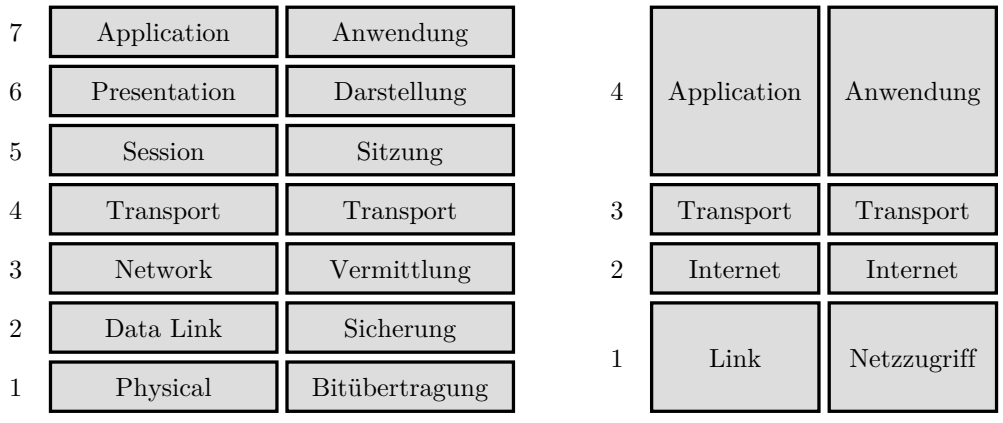
2.2.1. ISO/OSI- und TCP/IP-Referenzmodell

Das erste Schichtenmodell für Netzwerke wurde in den 1960er Jahren von der DARPA (Defense Advanced Research Projects Agency) für das amerikanischen Verteidigungsministerium entwickelt. Das sogenannte ARPANET bildete einen Vorläufer des heutigen Internet, aus dessen Schichtenmodell sich in den 1970er Jahren das TCP/IP-Referenzmodell entwickelte, das in Abbildung 2.5 (b) dargestellt ist. Parallel wurde von der ITU (International Telecommunication Union) ab dem Ende der 1970er Jahre das ISO/OSI Referenzmodell entwickelt, das in Abbildung 2.5 (a) dargestellt ist. Beide Schichtenmodelle beinhalten die gleiche Funktionalität, wobei die unterschiedlichen Funktionen lediglich anders gruppiert und zusammengefasst wurden. Die Entwicklung des TCP/IP-Referenzmodells begann zwar bereits in den 1970er Jahren, eine weite Verbreitung fanden die Protokolle TCP/IP jedoch erst in den 1990er Jahren mit dem Durchbruch des World Wide Web, so dass sich TCP/IP ab diesem Zeitpunkt gegen die konkurrierenden Protokolle IPX/SPX von Novell, AppleTalk von Apple und NetBEUI von Microsoft durchsetzen konnte.

Im Folgenden wird auf die Funktionen der in Abbildung 2.5 dargestellten Schichten eingegangen. Das ISO/OSI-Schichtenmodell teilt die Kommunikation hierbei in insgesamt sieben Schichten ein: Die erste Schicht stellt die sogenannte Bitübertragungsschicht dar, die für die Kanalkodierung auf dem verwendeten Medium (z.B. Kupferdraht oder Glasfaser) zuständig ist. Hierauf baut die Sicherungsschicht auf, die den abwechselnden Zugriff von zwei oder mehr Kom-

munikationspartnern regelt, die sich ein Medium teilen. Die Vermittlungsschicht ist für die Weiterleitung der Daten von einer Station über Zwischenstationen zur Zielstation zuständig und kann mehrere Netzwerke miteinander verbinden. Die Transportschicht etabliert eine Ende-zu-Ende Beziehung zwischen zwei Prozessen auf unterschiedlichen Rechnersystemen und kann z.B. Mechanismen zur Flusskontrolle, Reihenfolgeerhaltung oder zur Behandlung von Nachrichtenverlust beinhalten. Die Sitzungsschicht stellt eine weitere Abstraktionsebene zur Verfügung, die den Datenstrom zwischen Sender und Empfänger unabhängig von einer konkreten Transportschicht-Verbindung macht und somit robust gegenüber Verbindungsabbrüchen auf Transportebene werden lässt. Die Darstellungsschicht dient der effizienten systemunabhängigen Darstellung der übermittelten Daten und erlaubt somit z.B. den Datenaustausch zwischen unterschiedlichen Betriebssystemen und Hardwareplattformen, bei denen die Daten automatisch von einer Darstellungsart in eine andere überführt werden. Die oberste Schicht stellt die Anwendungsschicht dar, welche die Schnittstelle zu den Anwendungsprogrammen bildet, die auf dem Computersystem laufen. Das TCP/IP-Referenzmodell hingegen fasst die Sicherungs- und Bitübertragungsschicht zu einer gemeinsamen Netzzugriffsschicht zusammen und vereint die oberen drei Schichten des ISO/OSI-Modells zu einer gemeinsamen Anwendungsschicht, die alle oder eine Teilmenge der Aufgaben dieser Schichten erfüllt.

Der Einsatz von Schichtenmodellen bietet viele Vorteile gegenüber der monolithischen Implementierung ohne Schichten: Durch die Trennung der Gesamtfunktionalität in mehrere, voneinander unabhängige Schichten kann die Implementierung in getrennten Modulen erfolgen, die sich in Ihrer Funktion ergänzen und aufeinander aufbauen können. Ein Modul, das die Funktionalität einer (oder mehrerer) Schicht(en) implementiert, wird in diesem Kontext als Protokoll bezeichnet. Durch die Aufspaltung in unterschiedliche Schichten wird



(a) ISO/OSI (b) TCP/IP

Abbildung 2.5.: Übersicht über das ISO/OSI-Schichtenmodell (a) und das TCP/IP-Referenzmodell (b), jeweils in der englischen Version (links) und der deutschen Übersetzung (rechts)

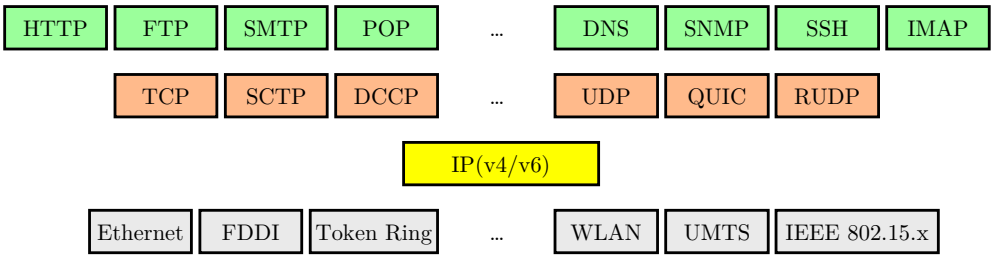


Abbildung 2.6.: Das Internet-Protokoll (IPv4/IPv6) ermöglicht den Informationsaustausch zwischen Netzwerken mit unterschiedlichen Netzzugriffsschichten

die Komplexität dieser Protokolle verringert und die Wartbarkeit erhöht, da jede Schicht einzeln implementiert und weiterentwickelt werden kann. Hierdurch können verschiedene Implementierungen derselben Schicht (und auch desselben Protokolls) für unterschiedliche Anwendungsgebiete, Betriebssysteme und Hardwarearchitekturen existieren, die zusätzlich an unterschiedliche Anforderungen angepasst sein können - wie z.B. eine geringe Latenz, ein hoher Datendurchsatz oder ein geringer Speicherverbrauch. Die in einem Schichtenmodell verwendeten Protokolle werden darüber hinaus häufig standardisiert, so dass eine Interoperabilität der verschiedenen Implementierungen für unterschiedliche Betriebssysteme ermöglicht wird. Im Fall des TCP/IP-Referenzmodells wird die Standardisierung von der IETF (Internet Engineering Task Force) durchgeführt.

Da jede Schicht über eine wohl definierte Schnittstelle zu seiner darüber und darunterliegenden Schicht verfügt, können die Implementierungen der einzelnen Schichten leicht gegeneinander ausgetauscht werden, um das Schichtenmodell an das jeweilige Anwendungsszenario anzupassen. So kann im Fall des TCP/IP-Referenzmodells z.B. das Transportprotokoll UDP gegen TCP ausgetauscht werden, falls die Anwendung Mechanismen wie eine Stau- oder Flusskontrolle und Reihenfolgeerhaltung benötigt. Zusätzlich kann unterhalb der Vermittlungsschicht auf eine Reihe von unterschiedlichen Übertragungsarten wie z.B. Ethernet, WLAN oder Glasfaser zurückgegriffen werden, während eine eindeutige Adressierung aller Stationen im Netzwerk über das Internet-Protokoll sichergestellt wird. Dies ist in Abbildung 2.6 dargestellt und wird als Narrow-Waist bezeichnet. Der Begriff Narrow-Waist bezieht sich in diesem Fall auf die Eigenschaft von IP, eine Vielzahl von Transport- und Sicherungsprotokollen miteinander zu verbinden und so eine *enge Taille* im Protokollstapel zu bilden.

Die Implementierung der Schichten werden als Protokolle bezeichnet, während die Gesamtheit der Protokolle eines Schichtenmodells als Protokollstapel oder Protokoll-Stack bezeichnet werden.

2.2.2. Schichtenmodelle für Sensornetze

Drahtlose Sensornetze galten lange Zeit als ungeeignet für den Einsatz des TCP/IP-Referenzmodells. Dies wurde aus den starken Ressourcenbeschränkun-

gen abgeleitet [70], die im Widerspruch zu einem hierarchisch in Schichten eingeteilten Protokollstapels mit streng in ihrer Funktionalität gekapselten Schichten zu stehen schien. Aber auch andere Eigenschaften des TCP/IP-Referenzmodells und von Sensornetzen schienen inkompatibel zueinander [70]: Während in einem IP-Netzwerk jeder Host eine eigene Identität in Form einer IP-Adresse besitzt und mit anderen Hosts Ende-zu-Ende kommuniziert, wird in einem Sensornetz häufig datenzentrisch kommuniziert, wobei Daten bereits im Netzwerk aggregiert, gefiltert und verarbeitet werden können. Hierbei entfällt die Notwendigkeit einer eindeutigen Identifikation der Sensorknoten bzw. eines Absenders oder Empfängers - stattdessen wird der jeweilige Messwert über seinen Messzeitpunkt und -ort identifiziert und abgerufen, z.B. durch die Verwendung eines geographischen Routing-Protokolls.

Schichtenmodelle sind sehr gut dafür geeignet, beliebige Anwendungen über ein Netzwerk miteinander zu verbinden, da sie standardisierte Schnittstellen anbieten und durch das Austauschen von Schichten flexibel an unterschiedliche Anwendungsszenarien angepasst werden können. Für Sensornetze hingegen sind spezialisierte Lösungen typisch, bei denen sowohl die Hardware als auch die Software bereits durch den Einsatz bestimmter Sensoren und Protokolle an ein Anwendungsgebiet angepasst sind.

Aus diesen Gründen wurde der Einsatz des TCP/IP-Referenzmodells in Sensornetzen lange Zeit vernachlässigt und stattdessen eine Vielzahl von anwendungsspezifischen Protokollen entwickelt, die direkt auf die Funktionen der Funkschnittstelle zugreifen [71]. Trotz dieser Bedenken brachten die Vorteile von Schichtenmodellen (siehe Abschnitt 2.2) mehrere unterschiedliche Schichtenmodelle für drahtlose Sensornetze hervor, die im Folgenden vorgestellt werden. Den Anfang macht hierbei das TCP/IP-Referenzmodell in Form des IPv6-Protokolls und des 6LoWPAN-Adaptionsschicht, gefolgt von ZigBee und einem Überblick über weitere Standards am Ende dieses Abschnitts.

TCP/IP und 6LoWPAN

Seit den Anfängen des Internets in den 90er Jahren hat sich das Internet-Protokoll zu einem der am häufigsten eingesetzten Kommunikationsprotokolle entwickelt und viele konkurrierende Protokolle verdrängt (z.B. IPX/SPX [72], AppleTalk [73]), wodurch die nahtlose Kommunikation zwischen Computersystemen von unterschiedlichen Herstellern und mit unterschiedlichen Betriebssystemen weltweit ermöglicht wurde. Die Vorteile dieser nahtlosen Kommunikation wurden auch im Forschungsgebiet der drahtlosen Sensornetze erkannt, und so wurde zunächst der Einsatz des IPv4-Protokolls für Sensornetze untersucht. Aufgrund der Annahme, das TCP/IP-Referenzmodell sei nicht für den Einsatz in Sensornetzen geeignet, verwendeten einige Ansätze ein Gateway oder Proxy, das die Protokollkonvertierung zwischen IPv4 auf der einen Seite und den Sensornetz-spezifischen Protokollen auf der anderen Seite übernahm [37]. Es stellte sich jedoch die Frage, ob die Protokollkonvertierung an der Grenze zwischen Sensornetz und Internet notwendig ist.

Im Gegensatz zu IPv4 bietet IPv6 viele Funktionen, die seinen Einsatz in Sensornetzen erleichtern, z.B. die automatischen Konfigurations- und Organisationsmechanismen des Neighbor-Discovery-Protokolls, der ausreichend große Adressraum mit bis zu 340 Sextillionen Adressen ($3{,}4 * 10^{38} \approx 2^{128}$) sowie der modulare Aufbau der IPv6-Header. Auf der Basis dieser Annahmen begann ab dem Jahr 2005 in der Network Working Group der IETF die Entwicklung der 6LoWPAN-Adaptionsschicht (RFC 4944 [9]), die den Einsatz von IPv6 in Sensornetzen ermöglichen soll. Das Protokoll übernimmt hierbei zwei Hauptaufgaben: Die Kompression des Nachrichten-Headers und die Fragmentierung der IP-Pakete.

Der Header eines IPv6-Pakets besitzt eine minimale Länge von 40 Byte, die vor allem durch die Absender- und Empfängeradressen (jeweils 16 Byte) verursacht wird. Zusätzlich können noch mehrere optionale Extension-Header wie die Hop-by-Hop-Options (>8 Byte), oder der Routing-Header (>8 Byte) vorhanden sein sowie der Header des jeweiligen Transportprotokolls, wie z.B. UDP (8 Byte). Zusammen mit dem MAC-Header von IEEE 802.15.4, der eine Länge von 23 Byte (64-Bit MAC-Adressen) besitzt, stehen von der 127 Byte großen MTU der Sicherungsschicht nur noch weniger als 56 Byte für den Payload des IPv6-Pakets zur Verfügung. Um die Größe der Header zu verringern, komprimiert die 6LoWPAN-Schicht die in den Headern enthaltenen Adressen und Felder (z.B. Flow-Label und Traffic-Class) auf eine Gesamtlänge zwischen 6 und 25 Byte, so dass mehr als 50 % der Länge des Headers eingespart werden können. Bei dieser Kompression werden zustandslose Kompressionsmechanismen verwendet, bei denen z.B. das Flow-Label und die Traffic-Class per Konvention auf Null gesetzt und weggelassen werden oder bei denen die IPv6-Adressen (teilweise) aus den MAC-Adressen rekonstruiert werden. Zusätzlich können auch zustandsbehaftete Kompressionsmechanismen verwendet, bei denen z.B. das Netzwerk-Präfix des Absenders und Empfängers komprimiert wird.

Laut RFC 2460 (IPv6 [8]) müssen IPv6-Pakete eine Mindestlänge von 1280 Byte unterstützen, während IEEE 802.15.4 nur eine maximale Nachrichtenlänge von 127 Byte zur Verfügung stellt. Aus diesem Grund verwendet 6LoWPAN zusätzlich zur Header-Kompression einen Fragmentierungsmechanismus, der das Aufspalten eines IPv6-Pakets auf mehrere Frames erlaubt. Neben diesen beiden Hauptaufgaben beinhaltet 6LoWPAN z.B. noch Mechanismen zur Unterstützung von Mesh-Routing. Eine Übersicht über einige existierende IPv6/6LoWPAN-Implementierungen für die Betriebssysteme TinyOS, Contiki und iSense findet sich in Abschnitt 2.1.2.

ZigBee

Der ZigBee Protokollstapel wurde von der ZigBee Alliance, einem Zusammenschluss von über 230 Unternehmen wie Philips, Texas Instruments, Freescale und Comcast, entwickelt und ist seit 2003 als Industriestandard verfügbar. ZigBee setzt wie 6LoWPAN auf der 802.15.4-Sicherungs- und Bitübertragungsschicht auf und wurde speziell für ressourcenbeschränkte und selbstorganisierende Netz-

werke wie z.B. Sensornetze konzipiert. Der ZigBee-Standard beinhaltet nur zwei Schichten: Die Vermittlungsschicht und die Anwendungsschicht, die zusätzlich Sicherheitsfunktionen beinhaltet. Die Vermittlungsschicht unterstützt sowohl Stern, Baum als auch Mesh-Topologien, wobei unterschiedliche Geräteklassen existieren, die entweder direkt am Routing teilnehmen (ZigBee-Router) oder nur Endknoten im Netzwerk sein können (ZigBee-End-Device). Die Anwendungsschicht bietet Profile für verschiedene Anwendungsgebiete wie z.B. Unterhaltungselektronik (RF4CE) oder Lichtsteuerung (Philips Hue).

Als Erweiterung des ZigBee-Standards existiert seit 2009 ZigBee-IP, das IPv6 zusammen mit der 6LoWPAN-Adaptionsschicht und dem RPL-Routing-Protokoll verwendet. Auf diese Weise können die ZigBee-Profile direkt in einem bestehenden IPv6-Netzwerk genutzt werden.

Weitere Standards

Neben den oben genannten Schichtenmodellen und Standards existieren noch eine Reihe weiterer (teilweise proprietärer) Lösungen. Das prominenteste Beispiel ist Bluetooth, das durch den Bluetooth 4.0/Low Energy Standard auch für größere Mesh-Netzwerke geeignet ist, wohingegen frühere Bluetooth-Standards nur Netzwerke mit einer maximalen Größe von acht Knoten erlaubten (ein Koordinator mit bis zu sieben Netzteilnehmern). Die ursprünglich für industrielle Feldbusse konzipierte HART-Protokollfamilie kann als WirelessHART in Sensornetzen eingesetzt werden. WirelessHART setzt wie ZigBee auf IEEE 802.15.4 auf und wird zur drahtlosen Steuerung von Industrieanlagen verwendet. Der Z-Wave Standard hingegen spezifiziert eine eigen Bitübertragungs- Medienzugriffs und Vermittlungsschicht und unterstützt Mesh-Netzwerke mit bis zu 232 Knoten. Z-Wave wurde für das Anwendungsgebiet der Heimautomation konzipiert. Weitere Standards sind z.B. DASH7, EnOcean (Gebäudeautomatisierung) oder ANT (Sportgeräte wie z.B. Pulsgurte, Fitnessuhren oder Fahrradtrainer).

2.3. Remote-Procedure-Calls

Analog zu den in Abschnitt 2.2 vorgestellten Schichtenmodellen kann auch die Funktionalität einer Software in unterschiedliche Schichten oder Komponenten aufgeteilt werden. Diese werden in Abhängigkeit von der konzeptionellen Ebene häufig als Pakete, Klassen oder Prozeduren bezeichnet, wobei Prozeduren die kleinste Einheit bilden. In Abhängigkeit von der verwendeten Programmiersprache werden Prozeduren hierbei auch als Methoden oder Funktionen bezeichnet. Eine Prozedur kann als Unterprogramm aufgefasst werden, das eine spezifische Aufgabe erfüllt. Zu diesen Aufgaben kann z.B. das Verändern oder das Abfragen eines Zustandes gehören oder die Verarbeitung einer Eingabe. Durch die Aneinanderreihung und Verschachtelung von Prozeduren können komplexe Programme entstehen, welche durch ihren modularen Aufbau ähnlich wie ein Schichtenmodell eine hohe Wartbarkeit und Übersichtlichkeit ermöglichen.

Der Aufruf von Prozeduren ist jedoch auf einen Prozess oder Computer beschränkt, da sämtliche Prozeduren und Parameter über einen einheitlichen Adressraum angesprochen werden. Um das Konzept der Prozeduren auch über die Grenzen eines Adressraums hinweg zu erweitern, wurde bereits gegen Ende der 1970er Jahren [74] das Konzept der entfernten Prozeduraufrufe entwickelt. Hierfür wurde ab Mitte der 80er Jahre von Andrew Birrell und Bruce Nelson der Begriff RPC (Remote-Procedure-Calls) geprägt [75]. RPCs erweitern die Adressierung einer Funktion und ihrer Parameter vom Adressraum eines einzelnen Prozesses oder Computers auf ein Netzwerk, wie z.B. das Internet. Auf diese Weise können Programme auf Implementierungen zugreifen, die von einem anderen Programm auf einem anderen Computer implementiert werden. Beispiele für einige der bekanntesten RPC-Frameworks sind ONC/Sun RPC [76], CORBA [77] oder Java RMI [78].

Im RPC-Kontext werden zwei Teilnehmer unterschieden: Der Server, der eine Implementierung über das Netzwerk zur Verfügung stellt und der Client, der auf eine Implementierung zugreifen möchte. Um die Schnittstelle der Funktion für den Server und Client genau festzulegen, d.h. den Namen, die Liste der Parameter, deren Typ sowie den Typ des Rückgabewerts, existieren unterschiedliche Beschreibungssprachen, die als IDL (Interface Description Language) bezeichnet werden. Aus dieser IDL werden der Client-Stub und der Server-Skeleton generiert, die zur Verbindung der Implementierung auf dem Server mit dem Client verwendet werden. Der Client ruft die entsprechende Funktion des Client-Stubs auf, die von dem jeweiligen RPC-Framework über ein Netzwerk an den Server-Skeleton weitergeleitet wird. Dort wird die Funktion mit den übergebenen Parametern aufgerufen und der Rückgabewert an den Client-Stub zurückgeschickt.

Es existieren verschiedene Implementierungen von RPC-Frameworks für unterschiedliche Anwendungsgebiete. Für drahtlose Sensornetze haben sich unter anderem TinyRPC [79] und SpartanRPC [80] entwickelt, die in Abschnitt 3.4.2 genauer beschrieben werden. Da das Programm auf dem Client für die weitere Ausführung auf das Ergebnis des RPC-Aufrufs vom Server angewiesen ist, werden RPC-Aufrufe häufig synchron ausgeführt, d.h. die Ausführung des Client-Programms wird angehalten, bis die Antwort des Servers eingetroffen ist. Aufgrund der starken Ressourcenbeschränkungen und der drahtlosen Kommunikation in drahtlosen Sensornetzen werden RPC-Aufrufe in Sensornetzen jedoch als asynchrone Vorgänge betrachtet. Da Sensornetze häufig Anwendungen mit einem einzigen Prozess realisieren, wird so der Stillstand des gesamten Sensorknotens während eines RPC-Aufrufs verhindert [80].

3. Verteilte Protokollstapel

In diesem Kapitel wird das Konzept der verteilten Protokollstapel vorgestellt: Abschnitt 3.1 stellt zunächst die Motivation hinter diesem Ansatz vor, während die allgemeine Funktionsweise in Abschnitt 3.2 beschrieben wird. Neben dem Konzept der verteilten Protokollstapel existieren alternative Ansätze, die in Abschnitt 3.3 vorgestellt werden, gefolgt von einem Überblick über verwandte Arbeiten in Abschnitt 3.4.

3.1. Ausgangslage

Der Einsatz von Schichtenmodellen in Sensornetzen bietet eine Vielzahl von Vorteilen (siehe Abschnitt 2.2.2) und wird von vielen Stellen vorangetrieben, so dass mittlerweile verschiedene Implementierungen von Schichtenmodellen für eine Vielzahl von Betriebssystemen zur Verfügung stehen (siehe Abschnitt 2.1.2). Ein limitierender Faktor, der den Einsatz von Schichtenmodellen in Sensornetzen in der Vergangenheit verhindert oder eingeschränkt hat, sind die Anforderungen einer solchen Implementierung in Zusammenhang mit den Ressourcenbeschränkungen von Sensornetzen [70].

Diese Erfahrung musste auch der Autor dieser Arbeit machen, als er in Zusammenarbeit mit der coalesenses GmbH im Herbst 2009 die Betreuung einer Diplomarbeit übernahm, welche die Implementierung eines IPv6-Stack für die iSense-Sensorknotenplattform zum Ziel hatte. Die im Rahmen dieser Diplomarbeit entwickelte Implementierung ist auf den Sensorknoten vom Typ JN5148 (128 kB Programmspeicher) lauffähig, während die Implementierung auf der vorherigen Generation dieser Hardwareplattform, dem JN5139 (96 kB Programmspeicher), nicht lauffähig ist. Der Grund für diese Einschränkung liegt in der Menge des auf diesen Plattformen zur Verfügung stehenden Programmspeichers, der beim JN5139 nicht ausreicht, um das iSense-Betriebssystem zusammen mit dem IPv6-Stack zu betreiben. Aber auch auf dem JN5148 beansprucht der IPv6-Stack einen großen Anteil des zur Verfügung stehenden Speichers.

Dieses Phänomen ist nicht auf das iSense-Betriebssystem beschränkt, sondern betrifft z.B. auch das Contiki-Betriebssystem: Tabelle 3.1 zeigt die Größe der Implementierung des IPv6-Stack des iSense- und des Contiki-Betriebssystems sowie dessen Anteil am insgesamt auf dieser Hardwareplattform zur Verfügung stehenden Programmspeichers. Für eine bessere Vergleichbarkeit wurde neben Contiki für die MSP430-Plattform auch die Jennisense[1] Portierung des

[1] https://github.com/teco-kit/Jennisense

Betriebssystem	Contiki	Contiki	Contiki	iSense	iSense
Plattform	MSP430	JN5139	JN5148	JN5139	JN5148
Speicher	48 kB	96 kB	128 kB	96 kB	128 kB
6LoWPAN	4,6 kB	8,0 kB	5,9 kB	7,8 kB	5,1 kB
IPv6	7,4 kB	12,3 kB	8,7 kB	31,6 kB	18,3 kB
ND	6,8 kB	13,3 kB	9,3 kB	7,6 kB	5,0 kB
ICMP	0,8 kB	1,3 kB	1,0 kB	2,1 kB	1,2 kB
UDP	0,7 kB	0,3 kB	0,2 kB	2,1 kB	1,1 kB
TCP	2,3 kB	5,1 kB	3,7 kB	N/A	N/A
Routing	9,0 kB	12,9 kB	8,4 kB	N/A	N/A
$\sum$	31,6 kB	53,2 kB	37,2 kB	51,2 kB	30,6 kB
% des Speichers	65,8 %	55,4 %	29,1 %	53,3 %	23,9 %

Tabelle 3.1.: Größe der IPv6-Implementierungen von Contiki (SICSlowpan) und iSense (iSIPS) auf den Plattformen MSP430, JN5139 und JN5148 im Vergleich zum verfügbaren Programmspeicher

Contiki-Betriebssystems für den JN5139 und JN5148 verwendet, die von der TECO Gruppe am KIT (Karlsruhe Institute of Technology) entwickelt wurde. Wie in Tabelle 3.1 dargestellt ist, benötigen die Contiki- und iSense-IPv6-Implementierungen mehr als die Hälfte (etwa 53 bis 55 %) des insgesamt auf dieser Plattform verfügbaren Programmspeichers, während es auf dem JN5148 etwa 24 bis 29 % sind. Auf dem MSP430 benötigt die Contiki-IPv6-Implementierung sogar beinahe zwei Drittel (65,8 %) des verfügbaren Programmspeichers. Diese Zahlen verdeutlichen, dass die Speicherplatzanforderungen von IPv6 den Einsatz auf den stärker ressourcenbeschränkten Plattformen verhindern oder stark einschränken.

Um den Einsatz von IPv6 auf stärker ressourcenbeschränkten Geräten zu ermöglichen, können entweder die Speicherplatzanforderungen von IPv6 auf den Sensorknoten reduziert werden oder aber der zur Verfügung stehende Speicherplatz erhöht werden. Dies kann entweder den Betrieb von IPv6 auf der entsprechenden Plattform überhaupt erst ermöglichen (z.B. auf der JN5139-Plattform) oder die Integration zusätzlicher Protokolle und Mechanismen ermöglichen, die vorher aus Platzgründen nicht umsetzbar waren, wie z.B. das Routing-Protokoll RPL, das Anwendungsprotokoll CoAP oder die Sicherheitsprotokolle IPsec und SSL (zzgl. Schlüsselverteilung).

Die Möglichkeit, IPv6 ohne eine Erhöhung des verfügbaren Speicherplatzes auf stärker ressourcenbeschränkten Geräten einsetzen zu können, bietet hingegen auch finanzielle Vorteile: Abbildung 3.1 stellt die Entwicklung des verfügbaren Programmspeichers in Abhängigkeit von den Kosten des jeweiligen Mikrocontrollers für den MSP430 von Texas Instruments und den JN51xx von Jennic/NXP zwischen den Jahren 2011 und 2013 dar. Beide Teildiagramme zeigen, dass der größte Teil der Sensorknoten mit höchstens 128 kB Programmspeicher ausgestattet ist, während nur wenige Modelle mit mehr als 128 kB Programmspeicher existieren (<15 %). Hierbei ist zu erkennen, dass im Jahr 2011 ein Mikrocontroller mit 128 kB Programmspeicher erst ab einem Kaufpreis von US$ 3,30

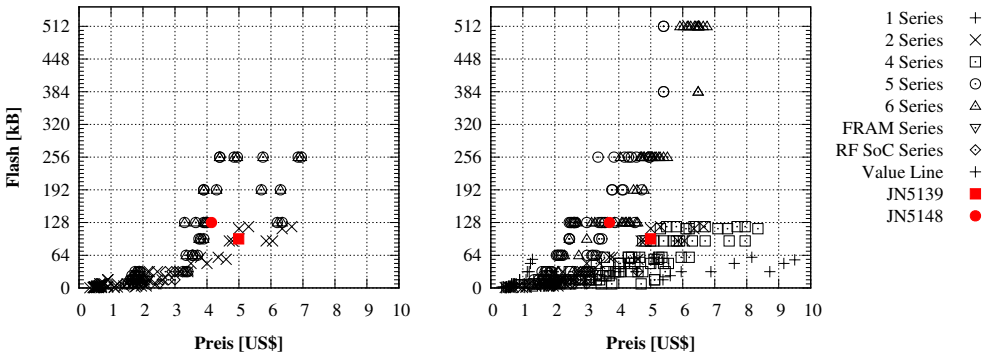


Abbildung 3.1.: Entwicklung der Kosten für Mikrocontroller in Abhängigkeit von der Flash-Speichergröße am Beispiel des MSP430 (schwarz) und JN51xx (rot) zwischen 08/2011 (links) und 9/2013 (rechts) (Quelle: Texas Instruments [81] und DigiKey [82]). Der Großteil der Sensorknoten ist mit maximal 128 kB Programmspeicher ausgestattet (>85 %)

verfügbar war. Aufgrund der fortschreitenden Entwicklung der Halbleitertechnologie konnte dieser Preis auf etwa US$ 2,45 im Jahr 2013 reduziert werden. Dies gilt jedoch nur für einige wenige Modelle, und aus der Gesamtverteilung in Abbildung 3.1 ist zu erkennen, dass in Abhängigkeit von den gewünschten Funktionen Preise zwischen US$ 5 und US$ 6 realistisch sind. Neben der Speicherausstattung wird der Preis eines Sensorknoten zudem von seiner Energieeffizienz und seiner Größe bestimmt. Ein Beispiel hierfür ist die FRAM Serie des MSP430, die mit besonders sparsamen ferroelektrischem Speicher ausgestattet sind. Modelle der FRAM Serie waren 2011 noch nicht verfügbar, während die maximale Speicherausstattung im Jahr 2013 lediglich 64 kB beträgt.

Obwohl die Halbleitertechnologie vermutlich auch weiterhin Fortschritte machen und auch zukünftig für ein Sinken der Kosten der Sensorknoten sorgen wird, kann durch die Nutzung intelligenter Algorithmen die Kostenverringerung weiter beschleunigt werden. Darüber hinaus ist es nicht immer wünschenswert, neue Sensorknoten für ein neues Anwendungsgebiet anschaffen zu müssen - so sind im WISEBED-Testbed an der Universität zu Lübeck insgesamt etwa 100 Sensorknoten verfügbar, wovon lediglich 18 vom Typ JN5148 sind und somit den Betrieb des iSense-IP-Stack ermöglichen, wohingegen über 85 Sensorknoten vom Typ MSP430 und JN5139 im Testbed vertreten sind.

Die vorliegende Arbeit bietet eine Lösung für dieses Problem an, welche die Kooperation zwischen benachbarten Sensorknoten verwendet. Diese Lösung wird als verteilte Protokollstapel bezeichnet und im folgenden Abschnitt vorgestellt. Neben der in dieser Arbeit vorgestellten Methode existieren noch weitere Lösungsansätze, die in Abschnitt 3.3 zusammengefasst sind.

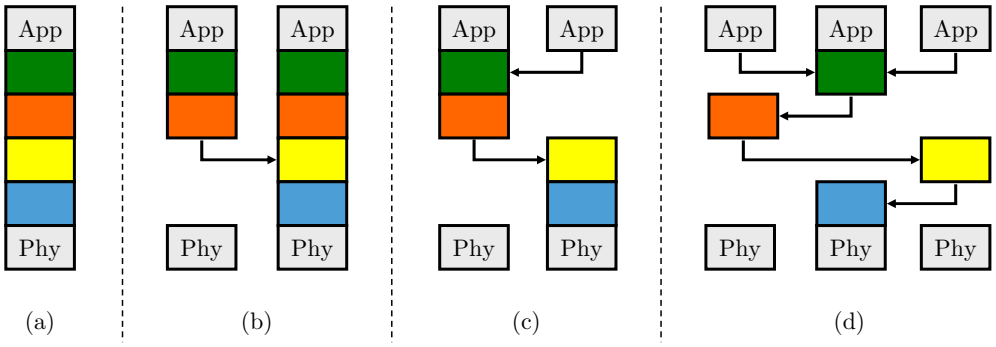


Abbildung 3.2.: Beispiele für die Verteilung der Protokollschichten des Protokollstapels auf mehrere benachbarte Sensorknoten

3.2. Funktionsweise

Eine der grundlegenden Eigenschaften drahtloser Sensornetze ist ihre Ressourcenbeschränkung: Ein einzelner Sensorknoten besitzt nur wenig Speicher, Rechenkapazität und Funkreichweite (siehe Abschnitt 2.1.4). Aus diesem Grund verwenden eine Vielzahl der Protokolle und Algorithmen, die für Sensornetze konzipiert wurden, die Kooperation zwischen benachbarten Sensorknoten, um diese Ressourcenbeschränkungen zu überwinden (siehe Abschnitt 2.1.5). Kooperation ermöglicht es, die beschränkten Ressourcen der einzelnen Sensorknoten miteinander zu kombinieren, um auf diese Weise komplexe Aufgaben zu erledigen, die von einem einzelnen Sensorknoten nicht erfüllt werden könnten. Diese Problemlösungsstrategie soll in der vorliegenden Arbeit die Verwendung von Protokollstapeln auf stärker ressourcenbeschränkten Geräten ermöglichen.

Im vorangegangenen Abschnitt wurde am Beispiel der IPv6-Implementierung des Contiki- und iSense-Betriebssystems gezeigt, dass der Betrieb dieses Protokollstapels auf stärker ressourcenbeschränkten Geräten nicht möglich ist, da nicht genug Programmspeicher auf den Knoten zur Verfügung steht, um alle Schichten des Protokollstapels zu implementieren. Die vorliegende Arbeit verfolgt daher den Ansatz, die Implementierung der Schichten des Protokollstapels auf mehrere Knoten zu verteilen. Bei dieser Vorgehensweise implementiert jeder Sensorknoten lediglich eine Untermenge des Protokollstapels und verwendet die Implementierung der fehlenden Schichten von seinen Nachbarknoten, um den Protokollstapel zu vervollständigen. Dies ist in Abbildung 3.2 dargestellt, die unterschiedliche Ausprägungen dieses Ansatzes beinhaltet: In Abbildung 3.2 (a) ist der Fall dargestellt, in dem alle Schichten des Protokollstapels von einem Sensorknoten implementiert werden. Die einzelnen Farben stehen hierbei für die unterschiedlichen Schichten des jeweiligen Protokollstapels - im Fall des TCP/IP-Referenzmodells könnten dies z.B. CoAP (grün), UDP (orange), IPv6 (gelb) und 6LoWPAN (blau) sein.

Diese Schichten können nun auf unterschiedliche Art und Weise auf zwei oder mehrere Sensorknoten verteilt werden, wobei die Anwendungslogik und die Sicherungs-/Bitübertragungsschicht (beide grau in Abbildung 3.2) auf jedem

Sensorknoten implementiert werden. Die Anwendungslogik wird auf jedem Sensorknoten benötigt, um die Aufgabe zu erfüllen, für die das Sensornetz konzipiert wurde, während die Sicherungs- und Bitübertragungsschicht häufig in Hardware auf dem entsprechenden Funkmodul implementiert werden und zur Kommunikation zwischen zwei Sensorknoten zwingend notwendig sind. Die restlichen Schichten können jedoch beliebig zwischen den Sensorknoten aufgeteilt werden.

Der einfachste Fall ist in Abbildung 3.2 (b) dargestellt: Ein Sensorknoten implementiert den vollständigen Protokollstapel (rechts), während ein weiterer Sensorknoten lediglich den oberen Teil des Protokollstapels implementiert (links). Eine Nachricht, die vom linken Sensorknoten an einen anderen Teilnehmer in Netz geschickt werden soll, durchquert folglich von der Anwendung aus zunächst lokal die beiden oberen Schichten des Protokollstapels, erhält die entsprechenden Protokoll-Header, und wird anschließend an die nächste Schicht auf einem benachbarten Sensorknoten weitergeleitet, der die weitere Verarbeitung der Nachricht übernimmt. Dieser Anwendungsfall der verteilten Protokollstapel ist für den Fall geeignet, dass einige Sensorknoten über ausreichend Programmspeicher verfügen, um den gesamten Protokollstapel zu implementieren, während einige nur einen Teil implementieren können. Die ist z.B. auf der iSense-Plattform der Fall und wird in Abschnitt 5.6 genauer beschrieben.

Ein weiterer Anwendungsfall ist in Abbildung 3.2 (c) dargestellt, bei dem beide Sensorknoten nur einen Teil des Protokollstapels implementieren. In diesem Beispiel implementiert der linke Sensorknoten lediglich die unteren beiden Schichten des Protokollstapels, während der rechte Sensorknoten die oberen beiden Schichten implementiert. Aus der Sicht des linken Sensorknotens ist die Vorgehensweise analog zu Abbildung 3.2 (a), der rechte Sensorknoten hingegen ist nun auf die Verwendung der oberen beiden Protokollschichten des linken Sensorknotens angewiesen. Die Anwendungslogik verwendet in diesem Fall folglich wie gewohnt die Schnittstelle der obersten (grünen) Protokollschicht, wobei die Aufrufe jedoch nicht lokal, sondern auf dem benachbarten Knoten ausgeführt werden. Dieser Anwendungsfall ist für ein Netzwerk von Sensorknoten geeignet, in denen keiner der Knoten über ausreichend Programmspeicher verfügt, um den gesamten Protokollstapel zu implementieren. Stattdessen implementiert jeder Sensorknoten etwa die Hälfte der Schichten.

Der Einsatz von verteilten Protokollstapeln ist nicht auf die Kooperation von zwei benachbarten Sensorknoten beschränkt, sondern kann auch weitere Knoten umfassen. Ein Beispiel hierfür ist in Abbildung 3.2 (d) dargestellt, das die Kooperation von insgesamt drei Sensorknoten zeigt. In diesem Beispiel implementieren die drei Knoten jeweils nur eine oder maximal zwei Schichten des Protokollstapels und verwenden die Implementierungen der übrigen Schichten der beiden benachbarten Sensorknoten. Ein Sensorknoten kann potenziell mehrere Rollen einnehmen: So greift der linke Knoten in Abbildung 3.2 (d) auf die Implementierung des grünen Protokolls des mittleren Sensorknotens zu und stellt gleichzeitig seine Implementierung des orangenen Protokolls für benachbarte Sensorknoten zur Verfügung.

Die Entscheidung, wie viele und welche Schichten auf welchen Sensorknoten implementiert werden, wird vom Anwendungsentwickler während der Konzeption des Sensornetzes in Abhängigkeit von den benötigten Protokollen und der verwendeten Hardware getroffen. Der Zugriff auf die Implementierung eines benachbarten Knotens erfolgt hierbei durch entfernte Prozeduraufrufe (Remote-Procedure-Calls, RPC): Ein Sensorknoten, der Protokoll A implementiert, stellt diese Implementierung den benachbarten Knoten zur Verfügung, indem er zusätzlich zu dem Protokoll eine RPC-Schnittstelle implementiert, die als Skeleton bezeichnet wird. Dieses Skeleton kann von benachbarten Sensorknoten über eine Schnittstelle verwendet werden, die als Stub bezeichnet wird. Stub und Skeleton werden hierbei mittels des in Abschnitt 4.1 vorgestellten Protokolls verbunden. Während der Betriebsdauer des Netzwerkes können die Sensorknoten selbstorganisiert darüber entscheiden, welche Implementierung auf welchem benachbarten Sensorknoten verwendet werden soll (siehe hierzu Abschnitt 4.1.1). Ein Sensorknoten kann prinzipiell beliebig viele Skeletons für die von ihm angebotenen Protokolle implementieren, wobei er für jedes Protokoll einen eigenen Skeleton benötigt. Umgekehrt kann ein Sensorknoten auch über unterschiedliche Stubs auf die Protokollimplementierungen mehrerer benachbarter Sensorknoten zugreifen.

RPC-Frameworks für klassische Computer, wie z.B. Sun RPC [76], CORBA [77] oder Java RMI [78] setzen oberhalb der Transportschicht auf den Protokollen TCP oder UDP auf und verwenden das Internetprotokoll, um Nachrichten zwischen unterschiedlichen Computern auszutauschen. Da das DPS-Protokoll die Verteilung eben dieses Protokollstapels auf mehrere benachbarte Sensorknoten vorsieht, muss ein alternatives Protokoll verwendet werden. Dieses Protokoll stellt lediglich die für das RPC-Framework benötigten Funktionen zur Verfügung und stellt somit eine schlanke Alternative zu den oben genannten RPC-Protokollen dar. Diese Vorgehensweise ist identisch zu dem von den beiden RPC-Frameworks TinyRPC oder SpartanRPC (siehe Abschnitt 3.4.2) verwendeten Ansatz, welche ebenfalls für den Einsatz in Sensornetzen konzipiert wurden.

3.3. Alternativen

Die Motivation hinter dem Einsatz von verteilten Protokollstapel ist es, dass die Implementierung eines Schichtenmodells, wie z.B. des TCP/IP-Referenzmodells, zwar mit vielen Vorteilen verbunden ist, jedoch auf stärker ressourcenbeschränkten Geräten nicht oder nur sehr eingeschränkt möglich ist (siehe Abschnitt 3.1). Dieses Problem kann durch die Verwendung der in dieser Arbeit vorgestellten verteilten Protokollstapel gelöst werden, es existieren jedoch noch weitere Lösungswege, die in diesem Abschnitt vorgestellt werden sollen.

Wie bereits in Abschnitt 3.1 erwähnt wurde, sind die Ressourcenbeschränkungen von Sensorknoten unter anderem abhängig von dem Preis, der für einen Sensorknoten gezahlt werden kann. Je mehr Geld für einen Sensorknoten ausgegeben wird, desto mehr Programmspeicher steht zur Verfügung. Bei der Anschaffung

eines Sensornetzes, das mehrere hundert oder sogar tausend Sensorknoten umfassen soll, spielt der Preis jedes einzelnen Sensorknotens jedoch eine große Rolle. Zusätzlich müssen die Anwendungsgebiete für drahtlose Sensornetze berücksichtigt werden, die in Abschnitt 2.1.3 vorgestellt wurden: Drahtlose Sensornetze werden häufig unter widrigen Umweltbedingungen eingesetzt, so dass mit dem Verlust oder der Zerstörung von Sensorknoten gerechnet werden muss. In einigen Fällen werden Sensorknoten sogar als Wegwerfprodukt betrachtet [83]. Es besteht folglich die Möglichkeit, die Ressourcenbeschränkungen durch den Kauf von mehr Programmspeicher zu verringern - dieser Ansatz ist jedoch aufgrund der häufig nur in begrenztem Umfang zur Verfügung stehenden finanziellen Mittel nicht uneingeschränkt durchführbar. Gleichzeitig konkurrieren weitere Bestandteile eines Sensorknotens um dieselben finanziellen Mittel, wie z.B. der Arbeitsspeicher, die Batterie oder die Qualität der Sensoren.

Eine weitere Möglichkeit besteht darin, den TCP/IP-Stack in ein spezialisiertes Hardwaremodul auszulagern. Dieser Ansatz wird z.B. von der Arduino-Plattform verfolgt, bei dem das Ethernet[2]-, GSM[3]- oder WLAN-Modul[4] neben der entsprechenden Netzwerkschnittstelle einen Mikrocontroller beinhaltet, der den TCP/IP-Stack implementiert und der Anwendung auf dem eigentlichen Arduino lediglich TCP- und UDP-Sockets zur Verfügung stellt. In diesem Fall wird der TCP/IP-Stack auf dem entsprechenden Hardwaremodul implementiert und vom Sensorknoten über eine API angesteuert. Durch die Verwendung eines solchen Hardwaremoduls steigen jedoch sowohl die Kosten als auch der Energieverbrauch. Darüber hinaus sind keine Anpassungen der IP-Stack-Implementierung möglich, wie z.B. die Wahl des Routing-Protokolls, der Einsatz eines Duty-Cycle-Protokolls oder die Integration von zusätzlichen Protokollen innerhalb des Stacks. Eine Korrektur von Fehlern oder die Integration von neuen Funktionen ist lediglich durch den Hersteller möglich und unter Umständen mit dem Neukauf des entsprechenden Moduls verbunden.

Neben einer Lösung durch zusätzliche Hardware ist auch eine Anpassung der Software möglich. Durch eine Optimierung der IPv6-Implementierung oder eine Anpassung/Reduktion des Funktionsumfangs auf die im jeweiligen Anwendungsfall benötigten Funktionen können die Anforderungen des IP-Stack an die zur Verfügung stehenden Ressourcen angepasst werden. Im Folgenden soll zunächst die Optimierung betrachtet werden, bevor im nächsten Absatz auf die Anpassung des Funktionsumfangs eingegangen wird. Die Optimierung der Implementierung ist zeitaufwändig und mit vielen Kompromissen verbunden: Soll die Programmgröße reduziert werden, geht dies entweder zu Lasten der Laufzeit oder zu Lasten der Wartbarkeit des Quellcodes, z.B. durch eine Umwandlung in eine monolithische Implementierung. Gleichzeitig sind hierbei keine großen Ersparnisse zu erwarten, wie der Vergleich der iSense-Implementierung mit der bereits optimierten Contiki-Implementierung in Tabelle 3.1 zeigt. Eine Reduktion des Funktionsumfangs hingegen kann die Programmgröße des IPv6-Stack

[2] http://arduino.cc/en/Main/ArduinoEthernetShield
[3] http://arduino.cc/en/Main/ArduinoGSMShield
[4] http://arduino.cc/en/Main/ArduinoWiFiShield

stark reduzieren, wie im Folgenden am Beispiel der Tiny COAP Sensors [84] Implementierung gezeigt werden soll:

Im Juli 2011 präsentierten einige Forscher von Ericsson Research eine CoAP-Implementierung, die in lediglich 48 Zeilen Assembler-Code (+160 Byte Nachrichtenspeicher) implementiert wurde, was etwa 200 Byte entspricht[5]. Diese Implementierung setzt keinen vollständigen und standardkonformen IPv6-Stack um, sondern unterstützt lediglich den Versand eines einzigen festgelegten CoAP-Pakets. Hierfür wurde das komplette Paket mit allen IPv6-, UDP- und CoAP-Headern fest im Programmcode kodiert. In diesem Paket werden vor dem Versand an mehreren bekannten Stellen der aktuelle Messwert, die Absenderadresse sowie die aktualisierte UDP-Prüfsumme eingefügt. Die Absenderadresse entspricht der link-lokalen Adresse, die aus der lokalen MAC-Adresse berechnet werden kann, während als Zieladresse eine fest codierte link-lokale Multicast-Adresse verwendet wird. Als Link-lokale Adressen werden die Adressen mit dem Präfix `fe80::/64` bezeichnet, die ausschließlich zur Kommunikation mit Knoten verwendet werden können, die sich auf demselben Link befinden - im Fall von IEEE 802.15.4 entspricht dies allen Knoten in Funkreichweite. Anschließend wird das Paket über eine Ethernet-Schnittstelle gesendet, woraufhin der Sensorknoten in einen Energiesparmodus versetzt wird und erst beim nächsten Sendeereignis wieder aufwacht. Diese Implementierung ist zwar sehr kompakt, bietet jedoch auch nur einen stark reduzierten und nicht standardkonformen Funktionsumfang an. Wie die Autoren selbst zugeben, verzichten sie auf den Duplicate-Address-Detection-Mechanismus, der laut RFC 4862 zwingend vorgeschrieben ist. Die Verwendung von link-lokalen Adressen führt dazu, dass die Kommunikation zu den Sensoren nur über den jeweiligen Link möglich ist, da link-lokale Adressen von Routern nicht weitergeleitet werden dürfen („Routers must not forward any packets with Link-Local source or destination addresses to other links.“, siehe [86]). Außerdem können die Sensorknoten mit dieser Implementierung nur eine einzige Funktion erfüllen und es existiert keinerlei Möglichkeit zur bidirektionalen Kommunikation - weder werden ICMP-Pakete (z.B. Echo-Request) beantwortet, noch können die Messwerte über CoAP nach Bedarf manuell oder mittels Conditional-Observe [87] abgerufen werden. Conditional-Observe erlaubt es einem Client, vom Server über die Änderung des Inhaltes einer CoAP Ressource informiert zu werden, sobald eine Änderung eintritt. Zusätzlich kann diese Änderung an Bedingungen geknüpft sein, wie z.B. „Informiere mich, sobald die Temperatur mehr als 30° C beträgt“.

Wie in Abschnitt 3.1 erläutert wurde, benötigen die vollständigen und standardkonformen IPv6-Stacks des Contiki- und iSense-Betriebssystems in Abhängigkeit von der verwendeten Hardwareplattform zwischen 30,6 und 53,2 kB Programmspeicher. Diese Anforderungen können im Extremfall durch eine starke Funktionsreduzierung und monolithische Implementierung auf die im vorangegangenen Absatz beschriebenen 48 Assemblerzeilen reduziert werden. Zwischen diesen beiden Extrema existieren viele Zwischenstufen, die in Abhängigkeit von den Anforderungen des jeweiligen Anwendungsszenarios die Programmgröße des

[5] ARM Cortex-M3 Prozessor und 32-bit Befehlen und einem Befehl pro Zeile, siehe [85]

IPv6-Stack reduzieren können. Da diese Anpassungen jedoch unmittelbar von diesen Anforderungen abhängig sind, existiert hierbei keine universelle Lösung. Stattdessen muss der Anwendungsentwickler diese Anpassungen während der Entwicklung der Sensornetzanwendung vornehmen.

Eine weitere Alternative bildet das Cross-Layer-Design, bei dem die strikte Trennung der einzelnen Schichten eines Schichtenmodells aufgehoben wird, um eine effizientere Implementierung und/oder eine höhere Leistung zu erreichen [88]. Beispiele hierfür finden sich z.B. in der Arbeit von [89], welche als Ziel die Optimierung des TCP-Datendurchsatzes über drahtlose Verbindungen verfolgt. Hierfür wird auf Informationen der Bitübertragungs- und Sicherungsschicht zurückgegriffen, welche in der Transportschicht verwendet werden. Obwohl diese Schichten im ISO/OSI- und TCP/IP-Schichtenmodell durch die Vermittlungsschicht getrennt sind, sieht der in [89] verfolgte Ansatz des Cross-Layer-Designs vor, dass trotzdem Informationen zwischen diesen Schichten ausgetauscht werden. Dies kann entweder durch gemeinsam genutzte Variablen oder Schnittstellen zwischen diesen Protokollen realisiert werden und wird auch als Cross-Layer-Optimization bezeichnet. Dieser Ansatz ist kompatibel zu dem in dieser Arbeit vorgestellten Konzept der verteilten Protokollstapel, da die hierbei entstehenden zusätzlichen Schnittstellen ebenfalls als RPC-Schnittstellen des DPS-Protokoll verwendet werden können. Die Leistung eines Protokollstapels lässt sich somit auch unter der Verwendung des DPS-Protokolls durch Cross-Layer-Design erhöhen. Ob die Kombination dieser beiden Ansätze sinnvoll ist und wie gut eine bestimmte Optimierung in diesem Zusammenhang funktioniert, hängt von dem jeweiligen Einsatzgebiet ab. An dieser Stelle sei jedoch angemerkt, dass die zusätzlichen Schnittstellen, welche durch das Cross-Layer-Design eingeführt werden, die Programmgröße der Protokollstapel weiter erhöhen und somit dem Ziel der verteilten Protokollstapel entgegenwirken. Auch im Cross-Layer-Design existieren jedoch Ansätze, mit denen die Programmgröße verringert werden kann, z.B. das „Merging of Adjacent Layers“ (vgl. [88]).

Es existieren auch kritische Meinungen zum Cross-Layer-Design, die z.B. in der Arbeit von [90] vorgestellt werden. Die Autoren von [90] argumentieren, dass Cross-Layer-Design die Übersichtlichkeit und die Wartbarkeit einer Architektur verringert und unvorhergesehene negative Wechselwirkungen auf den höheren Protokollschichten mit sich bringen kann. Dies wird am Beispiel des RBAR-Verfahrens [91] verdeutlicht, bei dem der Datendurchsatz in Abhängigkeit von der Signalqualität optimiert werden soll, indem das Modulationsverfahren der Bitübertragungsschicht durch die Sicherungsschicht angepasst wird. Wird das RBAR-Verfahren jedoch mit einem Routing-Protokoll kombiniert, das die Anzahl der Hops zwischen Start und Ziel minimiert, so wird der Ende-zu-Ende Datendurchsatz nicht wie gewollt erhöht, sondern sogar um einen Faktor zwischen 1,6 und 2,0 reduziert.

3.4. Verwandte Arbeiten

In diesem Abschnitt werden verwandte Arbeiten zu dem in dieser Arbeit vorgestellten Konzept der verteilten Protokollstapel vorgestellt. Im Gegensatz zu den im vorangegangenen Abschnitt vorgestellten alternativen Ansätzen verfolgen die in diesem Abschnitt vorgestellten Arbeiten einen vergleichbaren Ansatz, der sich jedoch in der Umsetzung unterscheidet. Diese Unterschiede werden für jede der vorgestellten Arbeiten einzeln beschrieben. Zunächst wird in Abschnitt 3.4.1 auf ein Framework eingegangen, das den Namen Distributed Protocol Stacks trägt und an der Universität Trient entwickelt wurde, jedoch eine andere Zielsetzung verfolgt. Anschließend werden in Abschnitt 3.4.2 einige RPC-Frameworks für Drahtlose Sensornetze vorgestellt, darunter TinyRPC und SpartanRPC. Den Abschluss bildet eine Vorstellung des Marionette Framework sowie das (sec)Fleck Betriebssystem für drahtlose Sensornetze in Abschnitt 3.4.3.

3.4.1. Distributed Protocol Stacks

In Ihrer Arbeit [92] stellen Kliazovisch und Granelli von der Universität Trient einen Ansatz vor, der die Implementierung einer Menge von Funktionskomponenten einer Protokollschicht von einem Netzwerkteilnehmer an eine andere Stelle im Netzwerk verschiebt, um die Leistungsfähigkeit des Netzwerks zu verbessern. Die Autoren aus [92] argumentieren, dass die Leistung von TCP/IP in drahtlosen Netzwerken schlecht ist, und geben als Gründe hierfür die mangelnde Zusammenarbeit und mangelnde Aktivität der einzelnen Schichten an. Ihrer Ansicht nach kann die Leistung von TCP/IP in drahtlosen Netzwerken durch die Kombination von Cross-Layer-Techniken [88] und Agenten-Basierten-Systemen [93] verbessert werden.

Ihr Ansatz, der als Distributed Protocol Stacks bezeichnet wird, erlaubt die

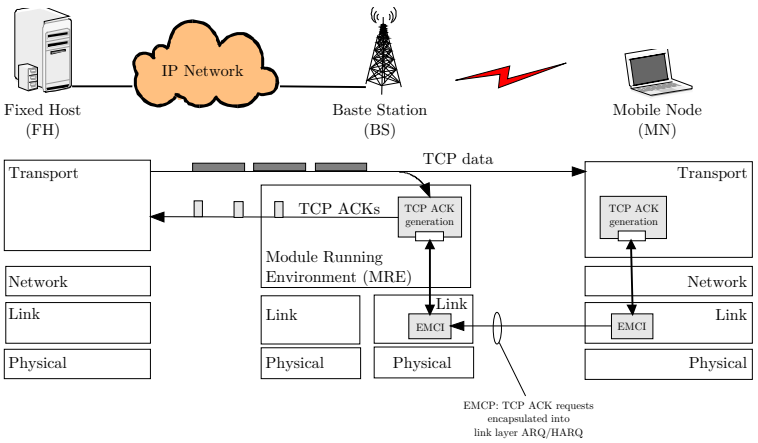


Abbildung 3.3.: Beispiel zu Distributed Protocol Stacks aus [92]: Verlagerung des Bestätigungsmechanismus von TCP aus dem drahtlosen Netzwerk auf eine Basisstation (Angepasste Version aus [92])

Verschiebung von atomaren Funktionen der Implementierung einzelner Protokollschichten im Netzwerk zu einer oder mehreren Basisstationen. Beispiele für solche atomare Funktionen sind der Bestätigungsmechanismus von TCP, Header-Kompressionsmechanismen, sowie Teile von IPsec oder der TCP-Staukontrolle. Die Autoren zeigen die Funktionsweise an einer Fallstudie, bei welcher der Bestätigungsmechanismus von TCP nicht auf den einzelnen Zielrechnern im drahtlosen Netzwerk ausgeführt wird, sondern stattdessen von einer Basisstation, die das drahtlose Netzwerk mit einem drahtgebundenen IP-Netzwerk verbindet.

Ein Beispiel hierfür ist in Abbildung 3.3 dargestellt: Ein Mobiler Knoten (MN) ist drahtlos über eine Basisstation (BS) mit einem drahtgebundenen IP-Netzwerk verbunden. Ein Host im IP-Netzwerk (FH) sendet TCP-Pakete an den mobilen Knoten. Diese werden mit TCP-Bestätigungen bestätigt, die jedoch nicht vom Empfänger der Pakete, sondern von der Basisstation erzeugt werden. Hierfür speichert die Basisstation die entsprechenden Bestätigungen beim Eintreffen der TCP-Pakete zunächst in einer lokalen Tabelle und wartet auf eine Empfangsbestätigung durch den mobilen Knoten. Diese Empfangsbestätigung wird auf der Sicherungsschicht (Link-Layer in Abbildung 3.3) mittels des im drahtlosen Netzwerk verwendeten (H)ARQ[6]-Protokolls versendet. Auf diese Weise können Mechanismen der Medienzugriffsschicht für die Leistungssteigerung der Transportschicht eingesetzt werden. Die Ergebnisse ihrer Simulationen in [94] zeigen, dass auf diese Weise die Latenz um 10 ms reduziert und der Datendurchsatz um 30-60 % gesteigert werden kann.

Die Arbeit von [92] hat das Ziel, eine oder mehrere atomare Funktionen der verwendeten Protokolle von den Teilnehmern eines Netzwerks zu einer Basisstation zu verschieben. Dies ist vergleichbar zu dem Konzept der verteilten Protokollstapel, wobei der in der vorliegenden Arbeit vorgestellte Ansatz nicht nur die Verschiebung einzelner atomarer Funktionen, sondern ganzer Protokolle im Netzwerk ermöglicht. Zusätzlich ist hierfür keine zentrale Komponente, wie z.B. einer Basisstation, notwendig. Stattdessen können die Protokolle im Netzwerk auf mehrere benachbarte Sensorknoten verteilt werden. Soll eine Basisstation die Aufgabe einer Protokollfunktion des gesamtes Netzwerkes übernehmen, so kann diese Basisstation zu einem Single-Point-of-Failure werden und die Skalierbarkeit des Netzwerkes einschränken.

3.4.2. TinyRPC und SpartanRPC

Die in dieser Arbeit vorgestellten verteilten Protokollstapel verwenden RPC-Mechanismen, um die Implementierung einer oder mehrerer Schichten von einem Sensorknoten auf benachbarte Sensorknoten verteilen zu können. RPC-Mechanismen wurden in drahtlosen Sensornetzen bereits für unterschiedliche Anwendungsfälle eingesetzt. Aus diesem Grund werden in diesem Abschnitt zwei bekannte RPC-Frameworks für drahtlose Sensornetze vorgestellt: TinyRPC, das erste RPC-Framework für drahtlose Sensornetze und SpartanRPC, das im Gegen-

[6] (Hybrid) Automatic-Repeat-reQuest-Protokoll, beinhaltet Bestätigungen in Kombination mit fehlerkorrigierenden Codes und wird z.B. von UMTS und LTE verwendet

satz zu TinyRPC Sicherheitsmechanismen beinhaltet, die zur Zugriffskontrolle verwendet werden können.

Das erste RPC-Framework für drahtlose Sensornetze wird in der Arbeit von [79] vorgestellt und wurde für das Betriebssystem TinyOS entwickelt. Dieses Framework, im Folgenden als TinyRPC bezeichnet, beinhaltet eine Erweiterung der Programmiersprache nesC, die den Aufruf von Funktionen auf benachbarten Sensorknoten erlaubt. TinyRPC übernimmt hierbei die Serialisierung und den Transport zwischen den Sensorknoten und beinhaltet Compilererweiterungen, welche die hierfür notwendigen Programmänderungen automatisch zur Übersetzungszeit vornimmt. Hierfür werden die Module, die TinyOS zur Kapselung von Funktionskomponenten verwendet, durch Remote-Module erweitert, die keine weiteren Anpassungen des Quellcodes durch den Anwendungsentwickler notwendig machen. TinyRPC unterstützt hierbei den Aufruf von entfernten Modulen sowohl per Unicast auf einem zufälligen benachbarten Knoten als auch per Broad- und Mulicast auf allen oder einigen der benachbarten Knoten. Neben diesem verbindungslosen Ansatz bietet TinyRPC auch einen verbindungsorientierten Ansatz, der von den Autoren als Discovery-Binding bezeichnet wird. Das Framework unterstützt hierbei das dynamische Binden von Knoten an die im Netzwerk angebotenen Dienste, so dass auf Knotenausfälle reagiert werden kann. Die Autoren evaluieren Ihren Ansatz auf insgesamt zehn TelosB-Knoten, die Sensorwerte mittels eines Clustering-Algorithmus an eine Datensenke weiterleiten. TinyRPC verwendet das GenericComm-Modul zum Versenden von Nachrichten und benötigt 10,7 kB Programmspeicher, wovon 9,3 kB auf GenericComm entfallen, das laut der Autoren von [79] von vielen TinyOS-Anwendungen ohnehin verwendet wird. Der RAM Overhead hingegen steigt im Vergleich zu einer manuell programmieren Version der Beispielanwendungen von ca. 400 Byte auf 900 Byte.

SpartanRPC [80] ist ein weiteres RPC-Framework für TinyOS und verwendet ebenfalls eine Erweiterung der nesC Programmiersprache. Im Gegensatz zu TinyRPC beinhaltet SpartanRPC Sicherheitsfunktionen, die eine Zugriffssteuerung der einzelnen RPC-Dienste erlaubt. Hierfür verwendet SpartanRPC ein Fähigkeiten-basiertes System, d.h. ein Sensorknoten kann einen RPC-Dienst nur genau dann nutzen, wenn er die hierfür benötigte Fähigkeit besitzt. Diese Fähigkeiten werden zur Übersetzungszeit statisch im Programmcode der Anbieter und Konsumenten von RPC-Diensten konfiguriert und entsprechen symmetrischen AES-Schlüsseln. Möchte ein Sensorknoten einen RPC-Dienst benutzten, der die Fähigkeit K voraussetzt, so muss er im Besitz des zugehörigen symmetrischen Schlüssels k sein und alle Anfragen an diesen Dienst mit einem 4 Byte MAC unter Verwendung dieses Schlüssels signieren. SpartanRPC verwendet darüber hinaus keinerlei Sicherheitsmechanismen wie z.B. Verschlüsselung oder einen Schutz vor Replay-Angriffen und unterstützt lediglich Nachrichten mit einer Maximallänge von 16 Byte, so dass nur eine einzige AES-Operation zur Generierung des MAC notwendig ist. Die Autoren zeigen die Funktionsfähigkeit von SpartanRPC anhand einer Implementierung des Directed-Diffusion-Algorithmus [95], der zur effizienten Verteilung von Informationen in einem Sensornetz verwendet

werden kann. Die Tests werden auf zwei Sensorknoten vom Typ Tmote Sky durchgeführt, wobei sich der Programmspeicherbedarf durch die Verwendung von SpartanRPC um 73-75 % und der Arbeitsspeicherbedarf um 61-74 % erhöht.

Sowohl TinyRPC als auch SpartanRPC stellen universelle RPC-Frameworks da, mit denen beliebige Programmfunktionen von einem Sensorknoten auf einem benachbarten Sensorknoten aufgerufen werden können. Beide Ansätze benötigen die Programmiersprache nesC und das Betriebssystem TinyOS und einige der von diesem angebotenen Dienste (z.B. GenericComm im Fall von TinyRPC). Beide Ansätze sind für den Austausch von kurzen RPC-Nachrichten konzipiert worden, die z.B. das Anschalten einer LED auf einem benachbarten Sensorknoten auslösen. Aus diesem Grund ist die maximale Nachrichtenlänge von SpartanRPC und TinyRPC auf 20 Byte begrenzt. Beide Frameworks unterstützen keine Fragmentierung von Nachrichten, so dass ein Austausch von IPv6-Paketen nicht möglich ist, da diese eine MTU von 1500 Byte voraussetzen. TinyRPC verwendet keine Warteschlange beim Senden oder Empfangen von Nachrichten, so dass immer nur genau ein Request gleichzeitig verarbeitet werden kann. Die verteilten Protokollstapel aus der vorliegenden Arbeit hingegen verwenden einen Fragmentierungsmechanismus und eine Warteschlange, so dass mehrere Nachrichten mit einer Länge von bis zu 2200 Byte gleichzeitig gesendet und empfangen werden können. TinyRPC und SpartanRPC erweitern den Syntax der Programmiersprache nesC um zusätzliche Funktionen, so dass eine Anpassung des Compilers notwendig ist und eine Verwendung unter anderen Programmiersprachen zunächst eine Portierung dieser Syntaxerweiterungen notwendig macht. Das Konzept der verteilten Protokollstapel kommt hingegen ohne eine Syntaxerweiterung und eine Anpassung des Compilers aus. Im Gegensatz zu TinyRPC verwendet das in der vorliegenden Arbeit vorgestellte Protokoll zudem Sicherheitsmechanismen, welche die Integrität und Authentizität der versendeten Nachrichten sicherstellen können. SpartanRPC bietet ähnliche Mechanismen, bietet jedoch keinen Schutz vor Replay-Angriffen.

3.4.3. Marionette und (sec)Fleck

Im Gegensatz zu den allgemeinen RPC-Frameworks im vorangegangenen Abschnitt wurden auch spezialisierte Algorithmen entwickelt, die RPC zur Fernsteuerung oder zum Debugging von Sensornetzen einsetzen. In diesem Abschnitt sollen deshalb das Framework Marionette und das (sec)Fleck Betriebssystem vorgestellt werden. Marionette bietet die Möglichkeit, Sensorknoten von einem PC aus fernzusteuern, während (sec)Fleck die Verwendung von RPC-Diensten auf benachbarten Sensorknoten ermöglicht.

Marionette

Die Entwicklung und das Debugging von Sensornetzanwendungen stellt aufgrund der verteilten Natur und der eingeschränkten Ausgabefunktionen von Sensornetzen eine große Herausforderung dar. Neben der Funkschnittstelle verfügen

Sensorknoten üblicherweise lediglich über eine (oder wenige) LEDs, die zur Mitteilung des internen Zustands oder zur Signalisierung von Ereignissen verwendet werden können. Bei der Entwicklung eines Sensornetzes mit mehreren hundert oder sogar tausend Sensorknoten kann die blinkende LED eines Sensorknotens jedoch leicht übersehen werden.

Um bei der Entwicklung und dem Debugging von Sensornetzanwendungen zu helfen, wird in der Arbeit von [96] das RPC-Framework Marionette vorgestellt. Marionette bietet die Möglichkeit, jeden einzelnen Sensorknoten in einem Netzwerk von einem PC aus vollständig zu überwachen und fernzusteuern. Hierfür können von einem PC aus Kommandos an die einzelnen Sensorknoten geschickt werden, die den Lese- und Schreibzugriff auf sämtliche Variablen und Datenstrukturen des Sensorknotens ermöglichen (mittels *PEEK* und *POKE*). Neben diesen systemnahen Funktionen bietet Marionette auch den Zugriff auf beliebige Funktionen an, die z.B. das Senden oder den Empfang einer Nachricht über die Funkschnittstelle oder den Zugriff auf die Sensoren erlauben.

Hierfür verwendet Marionette einen Embedded-RPC (ERPC) genannten Ansatz, der zur Übersetzungszeit eine XML Datei erzeugt, welche die Speicheradressen sämtlicher Variablen, Datenstrukturen und Funktionen der Sensornetzanwendung beinhaltet (Funktionen müssen hierfür mit dem Schlüsselwort *@rpc()* versehen werden). Diese XML Datei kann dann von einem Python-basierten Client auf einem PC zur Steuerung des Sensornetzes verwendet werden. Auf diese Weise kann von einer zentralen Stelle aus das Verhalten des gesamten Sensornetzes überwacht und gesteuert werden. Die XML Datei wird auf dem Sensorknoten gespeichert, so dass ein PC einen Sensorknoten ohne Kenntnis der Programmlogik verwenden kann, indem er in einem ersten Schritt zunächst die XML-Datei vom Sensorknoten bezieht. Marionette beinhaltet keine Sicherheitsfunktionen, da die Autoren von [96] annehmen, dass Sensorknoten unter der zuverlässigen Kontrolle des Anwendungsentwicklers stehen. In ihrer Evaluation geben die Autoren von Marionette an, dass das Framework 3,6 kB Programmspeicher und 153 Byte Arbeitsspeicher zur Laufzeit benötigt. Zusätzlich werden etwa 20 kB zur Speicherung der XML-Datei auf den Knoten benötigt. Marionette verwendet die Routing-Protokolle Drip und Drain [97].

Marionette ermöglicht es, die Knoten von einem an das Sensornetz angeschlossenen PC fernzusteuern. Ähnlich wie bei dem Konzept der verteilten Protokollstapel stellen die Sensorknoten Bestandteile ihrer Implementierung über RPC zur Verfügung. Dies dient jedoch nicht der Kommunikation der Knoten untereinander, sondern dem Debugging und der Steuerung des Sensornetzes durch eine zentrale Komponente. Dies ist vergleichbar zu dem in dieser Arbeit in Abschnitt 5.7 vorgestellten zentralisierten Routing-Protokoll. Die Art der zur Verfügung gestellten Funktionen unterscheidet sich jedoch, da Marionette einen Zugriff auf die Variablen und Hardware des Sensorknoten zum Ziel hat, während der in dieser Arbeit vorgestellte Ansatz die Funktionen eines Kommunikationsprotokolls zur Verfügung stellt.

(sec)Fleck

Das Fleck Betriebssystem [98] für drahtlose Sensornetze (FOS = Fleck Operating System) wurde als Alternative zu den entweder ereignisorientierten (TinyOS) oder Multithreading-Betriebssystemen (Contiki) für Sensornetze entwickelt. Im Rahmen der vorliegenden Arbeit ist hierbei vor allem von Interesse, dass FOS RPCs verwendet, um Anwendungen den nahtlosen Zugriff auf Dienste zu ermöglichen, die entweder lokal oder auf benachbarten Sensorknoten implementiert sein können. Hierbei unterstützt FOS ähnliche Dienste wie Marionette (siehe oben), die z.B. den Lese- und Schreibzugriff auf den Speicher eines Knotens, das Lesen von Sensorwerten oder das Schalten der LEDs erlauben. Im Gegensatz zu Marionette werden diese Dienste jedoch nicht von einem PC verwendet, der über ein Multi-Hop-Routing-Protokoll mit den Sensorknoten im Netzwerk kommunizieren kann, sondern von benachbarten Sensorknoten im Netzwerk.

Die Autoren stellen mit secFleck [99] eine Erweiterung vor, die FOS um Sicherheitsfunktionen erweitert. secFleck verwendet ein Hardwaremodul zur Integration des RSA Public-Key-Verschlüsselungsverfahrens [100] und des symmetrischen XTEA-Algorithmus (eXtended Tiny Encryption Algorithm [101]), die zur Verschlüsselung und Authentifizierung der RPC-Funktionen des Fleck Betriebssystems verwendet werden und durch Sequenznummern zusätzlichen Schutz vor Replay-Angriffen bieten. Neben der Unicast-Kommunikation zwischen benachbarten Sensorknoten unterstützt secFleck auch die Erzeugung von Gruppenschlüsseln, so dass Gruppenkommunikation ermöglicht wird. Auf diese Weise kann ein Knoten einen Dienst auf mehreren Nachbarn gleichzeitig verwenden, um z.B. alle Temperaturwerte seiner Nachbarn abzurufen.

(sec)Fleck verfolgt ähnlich wie Marionette das Ziel, die grundlegenden Funktionen der Sensorknoten über RPC zur Verfügung zu stellen. Hierbei unterscheidet es sich von dem Ansatz der verteilten Protokollstapel, die den Zugriff auf die Funktionen von Kommunikationsprotokollen durch benachbarte Sensorknoten ermöglichen. Im Gegensatz zu Marionette verfolgt (sec)Fleck jedoch einen zu den verteilten Protokollstapeln vergleichbaren dezentralen Ansatz, bei dem die Sensorknoten im Netzwerk direkt miteinander kommunizieren können und so direkt auf die Implementierung der benachbarten Sensorknoten zugreifen können. (sec)Fleck wurde jedoch als Betriebssystem konzipiert, so dass eine Integration in bestehende Anwendungen aufwendig ist, während die verteilten Protokollstapel prinzipiell in beliebige Anwendungen integriert werden können.

4. Entwurf

Das vorliegende Kapitel beinhaltet eine Beschreibung des DPS-Protokolls. Hierbei wird in Abschnitt 4.1 zunächst der Protokollablauf beschrieben, der mit der Discovery- und Advertisement-Phase (siehe Abschnitt 4.1.1) und dem anschließenden Verbindungsaufbau (siehe Abschnitt 4.1.2) startet. Nach dem Aufbau einer Verbindung können über das DPS-Protokoll Nachrichten ausgetauscht werden, was in Abschnitt 4.1.3 beschrieben wird. Die hierbei verwendeten Mechanismen zur Erkennung der Nachrichtenverlusts und von Verbindungsabbrüchen werden in Abschnitt 4.1.4 und Abschnitt 4.1.5 beschrieben.

Das vom DPS-Protokoll verwendete Nachrichtenformat wird in Abschnitt 4.2 beschrieben, wobei auf den Aufbau und den Inhalt der einzelnen Nachrichten des DPS-Protokolls eingegangen wird. Anschließend folgt eine Beschreibung der vom DPS-Protokoll verwendeten Sicherheitsmechanismen in Abschnitt 4.3. Hierbei wird vor allem auf das Angreifermodell und die Angriffsarten eingegangen sowie eine Übersicht über die verwendeten Sicherheitsmechanismen gegeben.

Zusätzlich zu dem grundlegenden Protokollablauf existieren mehrere Erweiterungen des Protokolls, die in Abschnitt 4.3.4 beschrieben sind. In Abschnitt 4.3.5 werden abschließend einige der Randbedingungen und Anforderungen beschrieben, die beim Einsatz von verteilten Protokollstapeln beachtet werden müssen.

Das in diesem Kapitel vorgestellte Protokoll wurde erstmals in der Form eines Posters [102] auf der EWSN 2012 in Trient und im Rahmen eines Vortrags auf den FGSN 2012 an der TU Darmstadt [103] vorgestellt. Weitere Beschreibungen des Protokolls finden sich in den Publikationen auf der CPScom2012 in Besançon [104] und CPScom2013 in Peking [105]

4.1. Protokollablauf

Basierend auf der in Abschnitt 3.2 vorgestellten Grundidee hinter den verteilten Protokollstapeln beschreibt dieser Abschnitt den Ablauf des DPS-Protokolls, das in mehrere Phasen unterteilt ist, die der Reihe nach durchlaufen werden und bei Bedarf wiederholt werden können. Das Protokoll beginnt nach dem Einschalten eines Sensorknoten. Zunächst werden alle auf dem Sensorknoten vorhandenen DPS-Skeletons und -Stubs initialisiert. Anschließend durchlaufen diese unabhängig voneinander die einzelnen Phasen des DPS-Protokolls. Ein Sensorknoten übernimmt für die DPS-Skeletons die Rolle eines Servers, während er für die DPS-Stubs die Rolle des Clients übernimmt. Ein Sensorknoten kann hierbei prinzipiell mehrere Skeletons und Stubs implementieren

und somit für das DPS-Protokoll gleichzeitig sowohl Server als auch Client sein. Um die fehlenden Protokoll-Implementierungen für ihre DPS-Stubs zu finden, müssen Clients zunächst die verfügbaren DPS-Skeletons der Server in ihrer Nachbarschaft finden - hierzu dient die Discovery- und Advertisement-Phase, die in Abschnitt 4.1.1 beschrieben wird. Anschließend folgt der Verbindungsaufbau zwischen den beiden Kommunikationspartnern (siehe Abschnitt 4.1.2). Sobald eine Verbindung aufgebaut wurde, können der Client und der Server mit dem Austausch von RPC-Nachrichten beginnen, wie in Abschnitt 4.1.3 beschrieben wird. Zusätzlich zu den in diesen Abschnitten beschriebenen Abläufen werden noch weitere Mechanismen verwendet, von denen einige optional sind und in Abhängigkeit von der gewählten Anwendung eingesetzt werden können. Hierzu zählen unter anderem die Fragmentierung von Nachrichten sowie die Erkennung von Nachrichtenverlust und Verbindungsabbrüchen.

Neben dem in den einzelnen Abschnitten beschriebenen Nachrichteninhalt beinhaltet jede der Nachrichten einen DPS-Header, der die Nachricht als DPS-Nachricht identifiziert, den Nachrichtentyp festlegt und mehrere Bits beinhaltet, die dem Empfänger die Benutzung bestimmter Funktionen signalisieren (z.B. Fragmentierung oder Erkennung von Nachrichtenverlust). Dies wird in den entsprechenden Abschnitten genauer beschrieben. Zusätzlich können sämtliche Nachrichten einen optionalen Message-Authentication-Code beinhalten, der am Ende der Nachricht angehängt wird. Den Aufbau und die Funktionsweise dieses Codes wird in Abschnitt 4.3.3 genauer beschrieben.

4.1.1. Discovery und Advertisement

Um die DPS-Skeletons in seiner Funkreichweite zu finden, beginnt der Client Discovery-Nachrichten mittels Broadcast zu senden. Discovery-Nachrichten beinhalten die ID des Protokolls, für das ein Skeleton gefunden werden soll, und können des Weiteren eine Liste von Filtern beinhalten. Diese Filter können dazu verwendet werden, um nur die DPS-Skeletons von Servern zu finden, die bestimmte Eigenschaften aufweisen, z.B. nur Server mit einer permanenten Stromversorgung, einer wiederaufladbaren Energiequelle oder einer bestimmten Hard- oder Softwareplattform. Diese Vorgehensweise wird am Ende dieses Abschnitts anhand eines Beispiels verdeutlicht. Sobald ein Server eine Discovery-Nachricht erhält, prüft er zunächst, ob er das angefragte Protokoll in Form eines DPS-Skeleton implementiert und sämtliche in der Nachricht enthaltenen Filter auf ihn zutreffen. Falls dem so ist, antwortet er mit einer Advertise-Nachricht, die dieselbe Protokoll-ID und dieselben Filter beinhaltet wie die zugrunde liegende Discovery-Nachricht.

Der Client speichert sämtliche Advertise-Nachrichten, die er von den umliegenden Servern erhält, zusammen mit einer konfigurierbaren Metrik (z.B. Signalstärke oder -qualität) und sendet weiter Discovery-Nachrichten, bis er mindestens K_D Discovery-Nachrichten gesendet hat und mindestens eine Advertise-Nachricht empfangen hat. Durch die Wahl von K_D wird einerseits die Dauer der Discovery-Phase bestimmt und andererseits die Menge an potentiell empfang-

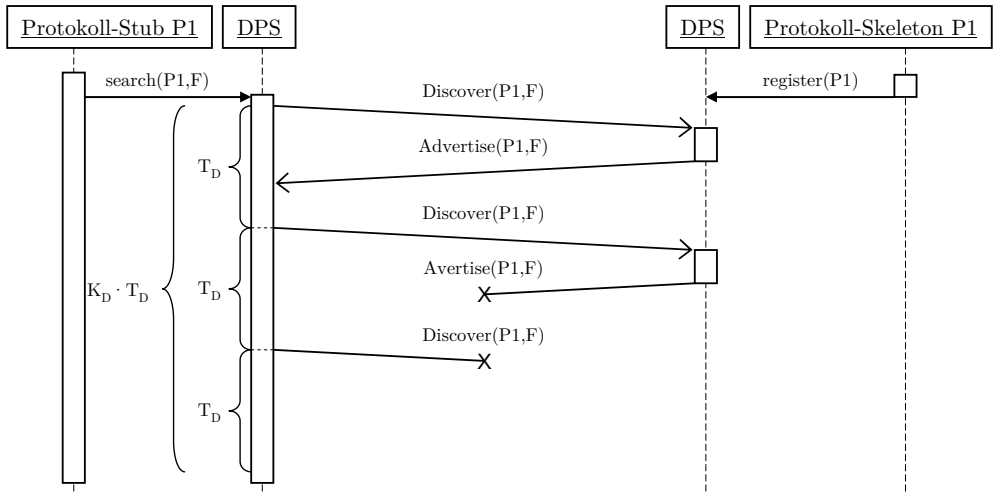


Abbildung 4.1.: Sequenzdiagramm des Nachrichtenaustauschs während des Discovery- und Advertisement-Mechanismus

baren Advertise-Nachrichten erhöht, da beide Nachrichtenarten in Abhängigkeit von der Signalstärke und -qualität auf dem Funkmedium verloren gehen können. Auf diese Weise soll sichergestellt werden, dass die Discovery-Nachrichten möglichst viele Nachbarn erreichen und die Advertise-Nachrichten von möglichst vielen Nachbarn empfangen werden können, um eine möglichst gute Auswahl des Servers zu ermöglichen.

Dieser Ablauf ist in Abbildung 4.1 als Sequenzdiagramm dargestellt und zeigt, wie der Protokoll-Stub auf dem DPS-Client und der Protokoll-Skeleton auf dem DPS-Server Discovery- und Advertise-Nachrichten austauschen. Die Wahl der Parameter dient hierbei lediglich als Beispiel - nähere Informationen hierzu finden sich im nächsten Kapitel in Abschnitt 5.1.1. Im vorliegenden Beispiel in Abbildung 4.1 sendet der Client insgesamt K_D=3 Discovery-Nachrichten im Abstand von jeweils T_D ms, wovon die ersten beiden Nachrichten beim Server ankommen, während die dritte Nachricht auf dem Funkmedium verloren geht. Der DPS-Server beantwortet die ersten beiden Discovery-Nachrichten mit jeweils einer Advertise-Nachricht, wovon nur die erste Nachricht beim Client ankommt, während die zweite Nachricht auf dem Funkmedium verloren geht.

Anschließend sortiert der Client alle empfangenen Advertise-Nachrichten anhand der gewählten Metrik und beginnt mit dem Verbindungsaufbau. Sollte der Client entweder keine Advertise-Nachricht empfangen haben oder aber keiner der versuchten Verbindungsaufbauten erfolgreich sein, so beginnt die Discovery-Phase erneut. In diesem Fall kann der Client sich jedoch dazu entscheiden, die angegebenen Filter zu reduzieren oder komplett auf diese zu verzichten.

Eine Übersicht der bisher definieren Filter ist in Tabelle 4.2 aufgelistet. Da die Filter als Byte-Wert in den Discovery-Nachrichten kodiert werden, stehen maximal 256 verschiedene Filter zur Verfügung, so dass neben den in Tabelle 4.2 dargestellten Filtern noch weitere definiert werden können. Die Liste ist in insgesamt drei Unterbereiche aufgeteilt: Der Bereich von 10 bis 19 gibt

	Filter	**Beschreibung**
1X	ENERGY_REMAINING	Noch verfügbare Energiereserven
10	10_PERCENT	Mindestens 10 %
11	20_PERCENT	Mindestens 20 %
...	...	
18	90_PERCENT	Mindestens 90 %
19	100_PERCENT	Vollständig geladen
2X	POWERED_	Art der Energieversorgung
20	BATTERY	Batterie
21	RECHARGEABLE	Wiederaufladbare Batterie (z.B. Solarzelle)
22	MAINS	Netzteil
3X	PLATTFORM_	Hardware-Plattform
30	JN5121	Jennic/NXP iSense JN5121
31	JN5139	Jennic/NXP iSense JN5139
32	JN5148	Jennic/NXP iSense JN5148
35	TELOSB	Crossbow TelosB (MSP430)
37	PACEMATE	MarathonNet Pacemate
38	VIRTUAL_NODE	Virtueller Sensorknoten (z.B. Simulator)

Abbildung 4.2.: Liste der unterstützten Filter

den verbleibenden Energievorrat des Servers an, während die Filter 20, 21 und 22 die Art der Energieversorgung angeben. Falls ein Sensorknoten ein Netzteil als Energieversorgung verwendet, erfüllt dieser Sensorknoten automatisch den Filtertyp 19 (Vollständig geladen). Der dritte Filterbereich gibt die verwendete Hardwareplattform an, wobei hier nicht nur physikalisch vorhandene Sensorknoten unterstützt werden, sondern auch simulierte Sensorknoten, die z.B. über Virtual-Links [106] angebunden sein können. Neben diesen drei Bereichen sind noch weitere Filter möglich, um ein bestimmtes Betriebssystem (z.B. iSense, TinyOS oder Contiki), Implementierung (z.B. iSense- oder Wiselib-Implementierung) oder weitere Eigenschaften des Servers beschreiben können (z.B. das Vorhandensein eines Kryptomoduls).

Beispiel

Gegeben seien die sechs Knoten in Abbildung 4.3: Zwei batteriebetriebene Clients (Knoten A und B), ein batteriebetriebener Server (Knoten C) und drei Server, die mit einem Stromanschluss ausgestattet sind (Knoten D, E und F). Die beiden Knoten A und B implementieren den Client-Stub des IPv6-Protokolls, während alle Server den zugehörigen Server-Skeleton implementieren. Im Folgenden wird lediglich der Discovery-Prozess des Knoten A betrachtet:

Nachdem Knoten A gestartet wurde, sendet er eine Discovery-Nachricht, um alle Server in seiner Umgebung zu finden, die den IPv6-Skeleton implementieren. Knoten A fügt der Discovery-Nachricht zusätzlich einen Filter hinzu, da er ausschließlich Antworten von Servern erhalten möchte, die über einen Stromanschluss verfügen. Diese Discovery-Nachricht wird von allen Knoten in der Funkreichweite von A empfangen (gestrichelter Kreis in Abbildung 4.3). Knoten F empfängt die Nachricht nicht, da er sich außerhalb der Funkreichweite befindet.

Anschließend antworten alle Server, die die Discovery-Nachricht empfangen haben und die den enthaltenen Filter erfüllen mit einer Advertise-Nachricht. Dies ist in Abbildung 4.4 dargestellt: Knoten B antwortet nicht mit einer Advertise-Nachricht, da es sich hierbei ebenfalls um einen DPS-Client handelt, während es sich bei Knoten C zwar um einen DPS-Server handelt, der jedoch batteriebetrieben ist und deshalb den Filter der Discovery-Nachricht nicht erfüllt. Knoten D und E antworten hingegen auf die empfangene Discovery-Nachricht, da es sich hierbei um DPS-Server handelt, die beide den angegebenen Filter erfüllen. Beide Knoten senden deshalb eine Advertise-Nachricht an Knoten A zurück. Knoten A misst die Signalqualität beim Empfang der beiden Advertise-Nachrichten und speichert diese.

Dieser Prozess wiederholt sich insgesamt K_D mal, wobei jedes Mal maximal Knoten D und E antworten. In Abhängigkeit vom Nachrichtenverlust hat Knoten A nach den K_D Durchläufen bis zu K_D Advertise-Nachrichten von Knoten D und bis zu K_D Advertise-Nachrichten von Knoten E empfangen. Wird der Parameter K_D zu niedrig gewählt, z.B. $K_D = 1$, so kann es sein, dass durch die zufällige Verteilung des Nachrichtenverlustes keine Advertise-Nachrichten von Knoten D empfangen werden, obwohl Knoten D eine potentiell bessere Metrik im Vergleich zu Knoten E aufweist und deshalb einen besseren Server für Knoten A darstellt.

4.1.2. Verbindungsaufbau und -abbau

Der Verbindungsaufbau wird in Form eines Drei-Wege-Handschlags durchgeführt und vom Client initiiert. Der Drei-Wege-Handschlag dient ähnlich wie bei TCP dazu, bestimmte Parameter zwischen den Kommunikationspartnern zu synchronisieren. Im Fall des DPS-Protokoll sind dies eine Nonce sowie die Paketzähler des Clients und des Servers, deren Verwendungszweck in den nachfolgenden Unterkapiteln beschrieben werden.

Nachdem sich der Client für einen Server entschieden hat, zu dem er eine Verbindung aufbauen möchte, löscht er die zugehörige Advertise-Nachricht aus seiner Liste und sendet eine Connect-Nachricht an diesen Server. Die Connect-Nachricht beinhaltet den Paketzähler der Clients sowie eine Nonce, die vom Client für diese Verbindung zufällig erzeugt werden. Der Server hat nun die Möglichkeit, die Connect-Nachricht des Servers mit einer Allow- oder einer Deny-Nachricht zu beantworten. Entschließt sich der Server dazu, den Verbindungsversuch anzunehmen, so sendet er eine Allow-Nachricht an den Client zurück. Entschließt

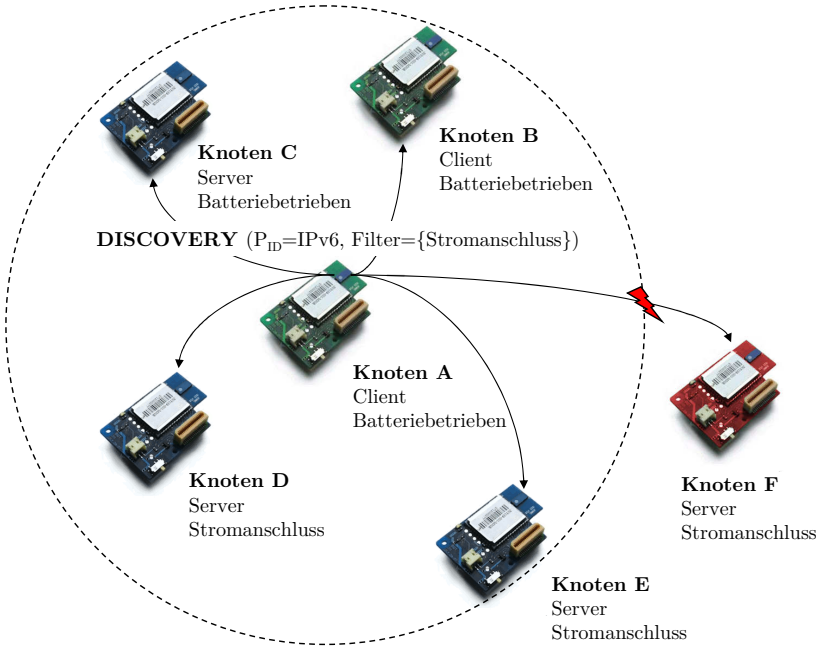


Abbildung 4.3.: Knoten A schickt eine Discovery-Nachricht, um die Server-Skeletons des IPv6-Protokolls in seiner Reichweite zu finden

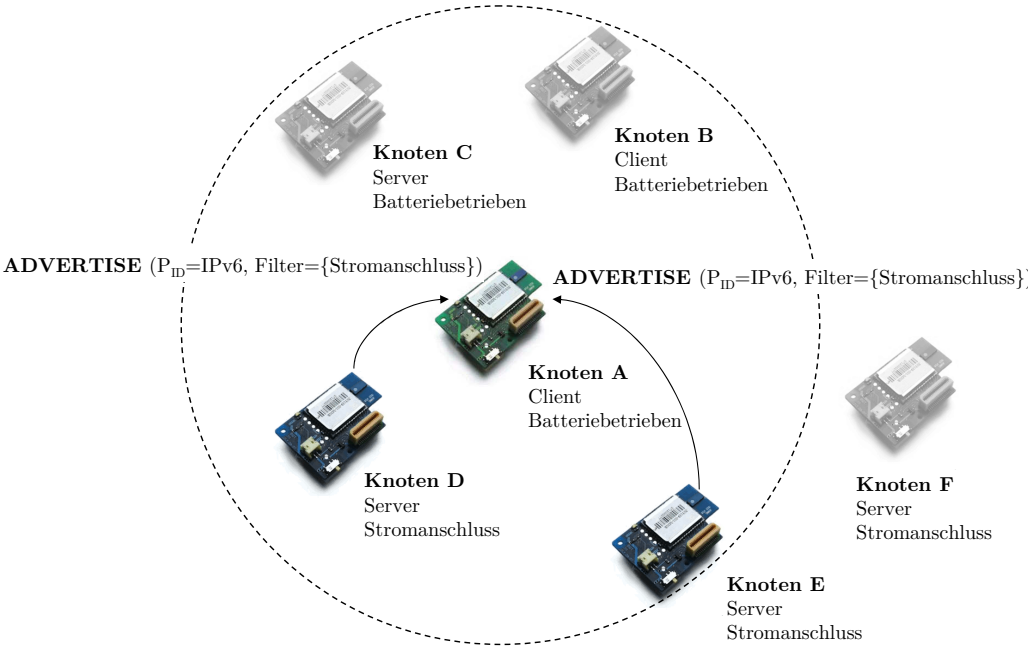


Abbildung 4.4.: Knoten D und E senden eine Advertise-Nachricht als Antwort auf die Discovery-Nachricht von Knoten A

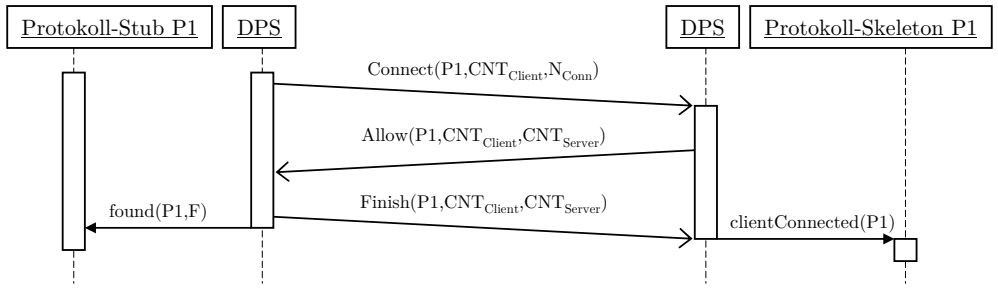


Abbildung 4.5.: Sequenzdiagramm des Verbindungsaufbaus

er sich hingegen, den Verbindungsversuch nicht anzunehmen, so sendet er eine Deny-Nachricht an den Client zurück. Beide Nachrichten beinhalten sowohl den Paketzähler des Clients, als auch einen vom Server erzeugten Paketzähler.

Empfängt ein Client eine Deny-Nachricht als Antwort auf die Connect-Nachricht, so beendet er den Verbindungsaufbau zu diesem Server und wählt eine neue Advertise-Nachricht aus der Liste aus. Sollten keine weiteren Nachrichten in der Liste vorhanden sein, beginnt er erneut mit der Discovery-Phase. Empfängt der Client hingegen eine Allow-Nachricht vom Server, so beendet er den Drei-Wege-Handschlag, indem er eine Finish-Nachricht an den Server sendet, die den Paketzähler des Clients und des Servers als Bestätigung beinhaltet. Auf diese Weise ist sichergestellt, dass sowohl der Client als auch der Server beide Paketzähler und die Nonce empfangen haben.

Sollte ein Client keine Antwort auf die Connect-Nachricht innerhalb eines festgelegten Zeitraumes erhalten, so wird er dieselbe Connect-Nachricht insgesamt k mal senden. Sollte auch nach dem letzten Sendevorgang keine Antwort innerhalb des festgelegten Zeitraumes eintreffen, so verfährt der Client genauso, als hätte er eine Deny-Nachricht erhalten. Sollte ein Server keine Antwort auf die Connect-Nachricht innerhalb eines festgelegten Zeitraumes erhalten, so wird er die Verbindung als halb-offen markieren. Halb-offene Verbindungen werden durch das Eintreffen einer beliebigen DPS-Nachricht als offen markiert, jedoch nicht anders behandelt.

Dieser Ablauf ist in Abbildung 4.5 als Sequenzdiagramm dargestellt und zeigt, wie der Protokoll-Stub auf dem DPS-Client nach der Discovery- und Advertise-Phase eine Verbindung zu dem Protokoll-Skeleton auf dem DPS-Server aufbaut. Hierfür sendet er eine Connect-Nachricht, die vom Server mit einer Allow-Nachricht beantwortet wird. Der Verbindungsaufbau wird mit der abschließenden Finish-Nachricht vom Client erfolgreich abgeschlossen, woraufhin der Protokoll-Stub und -Skeleton von DPS-Protokoll benachrichtigt werden. Hierbei werden der Paketzähler des Clients (CNT_{Client}) und des Servers (CNT_{Server}) sowie die Verbindungs-Nonce (N_{Conn}) ausgetauscht.

Der Verbindungsabbau folgt demselben Drei-Wege-Schema wie der Verbindungsaufbau, nur dass dieser sowohl vom Server als auch vom Client initiiert werden kann. Hierfür tauschen die Kommunikationspartner eine Disconnect-, Disconnect-Allow- und Disconnect-Finish-Nachricht nach dem oben beschriebenen Muster

aus. Ein Verbindungsabbau kann aus verschiedenen Gründen erfolgen, z.B. falls der entsprechende Knoten das Netzwerk verlassen muss (Veränderung der Position oder mangelnde Energie) oder die maximale Anzahl der verfügbaren DPS-Verbindungen für diesen Knoten erschöpft ist.

4.1.3. Nachrichtenaustausch

Nachdem eine DPS-Verbindung zwischen dem Server und dem Client aufgebaut wurde, kann mit dem Austausch von RPC-Nachrichten begonnen werden. Der Nachrichtenaustausch folgt hierbei einem Request-Response-Muster, wobei sowohl Client als auch Server eine Request-Nachricht an den jeweiligen Kommunikationspartner senden können, die dann mit einer Response-Nachricht beantwortet wird. Request-Nachrichten dienen dazu, eine Funktion des jeweiligen Kommunikationspartners aufzurufen, wobei die Parameter der Funktion als serialisierte Daten in der Nachricht enthalten sind. Die Response-Nachricht liefert das Ergebnis der Funktion in Form eines Rückgabewertes zurück. Beide RPC-Nachrichtenarten folgen demselben Aufbau und beinhalten einen Nachrichtenzähler, das Ziel des Aufrufs in Form der ID des Ziel-Protokolls und der ID der Funktion innerhalb des Protokolls, gefolgt von den serialisierten Funktionsparametern. Sowohl die Request- als auch die Response-Nachricht beinhalten den aktuellen Nachrichtenzähler des Senders der Request-Nachricht, um die Response-Nachricht eindeutig der Request-Nachricht zuordnen zu können. Die Serialisierung der Daten wird hierbei von der Implementierung des DPS-Stub und -Skeleton durchgeführt.

Da die Länge einer RPC-Nachricht durch die Serialisierung der Daten die maximale Paketgröße (MTU) der 802.15.4-Schicht übersteigen kann, unterstützt das DPS-Protokoll die Fragmentierung von RPC-Nachrichten. Sämtliche Fragmente der RPC-Nachricht beinhalten hierbei denselben Nachrichtenzähler, um die Fragmente einander eindeutig zuweisen zu können. Fragmentierte Nachrichten beinhalten einen zusätzlichen Fragmentation-Header, der einerseits die Gesamtlänge der Nachricht und andererseits die Position des aktuellen Fragments in der nicht fragmentierten Nachricht beinhalten. Auf diese Weise kann der Empfänger der Nachricht die einzelnen Fragmente in der korrekten Reihenfolge wieder zusammensetzen. Die Anwesenheit des Fragmentation-Headers wird durch das Setzen des F-Bits im Header der Nachricht signalisiert.

Ein Beispiel für den Nachrichtenaustausch zwischen Client und Server ist in Abbildung 4.6 (a) dargestellt. In diesem Beispiel wird eine Funktion des Protokoll-Stub aufgerufen, deren ID und Parameter als M_1 in serialisierter Form über das DPS-Protokoll als Request-Nachricht an den Server geschickt werden. Nachdem das DPS-Protokoll die Nachricht empfangen hat, wird der Inhalt M_1 vom Protokoll-Skeleton deserialisiert und ausgeführt. Die Antwort wird anschließend ebenfalls in serialisierter Form als M_2 über das DPS-Protokoll als Response-Nachricht zurück an den Client geschickt. Dort wird M_2 vom DPS-Protokoll empfangen, deserialisiert und vom Protokoll-Stub ausgewertet.

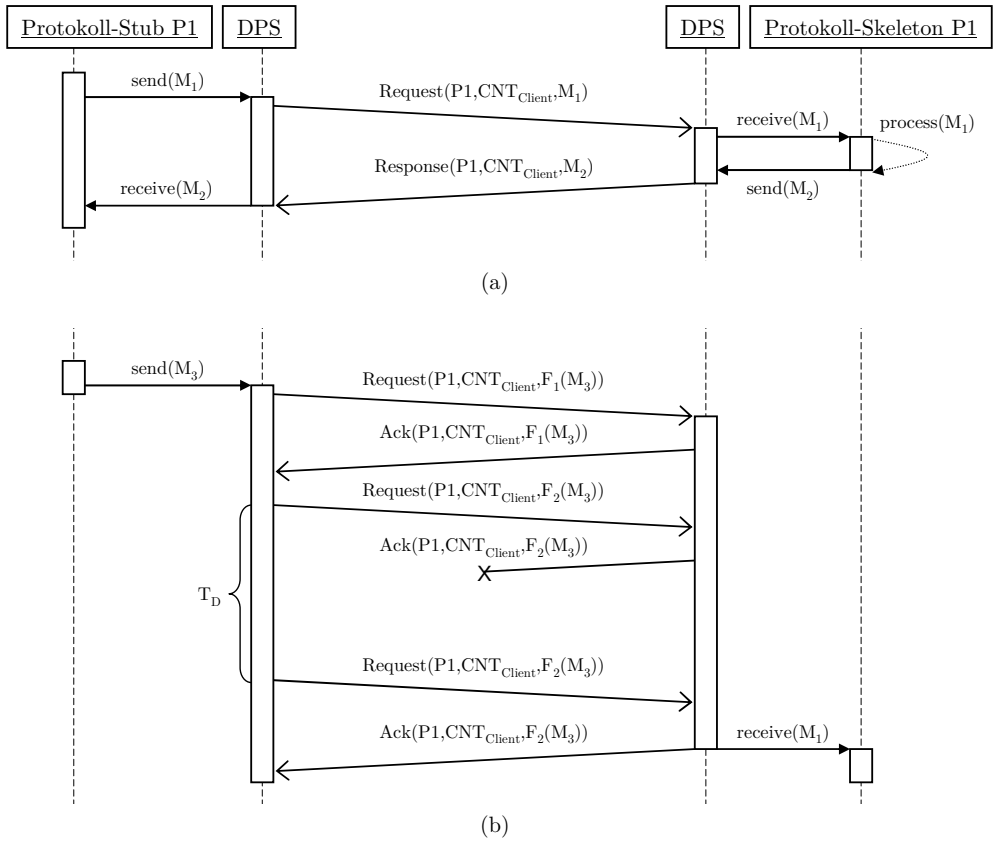


Abbildung 4.6.: Sequenzdiagramm des Nachrichtenaustauschs zwischen einem Client und Server ohne Bestätigungen (a) und mit Bestätigungen (b)

4.1.4. Erkennung von Nachrichtenverlust

Um sicherzustellen, dass alle Fragmente einer RPC-Nachricht beim Empfänger ankommen, unterstützt das DPS-Protokoll die Verwendung von Bestätigungen (Acks = Acknowledgements). Zusätzlich zur Erkennung von Nachrichtenverlust kann auf diese Weise die Reihenfolgeerhaltung der Nachrichten sichergestellt werden, die durch den Nachrichtenzähler eine totale Ordnung besitzen. Bestätigungen können vom Sender für beliebige Nachrichten angefordert werden und sind dadurch für alle Fragmente dieser Nachricht aktiviert. Dies wird durch das Setzen des eines Bits im Header der Nachricht signalisiert. Falls der Sender für eine Nachricht die Verwendung von Bestätigungen anfordert, so sendet er die einzelnen Fragmente der Nachricht einzeln und wartet zunächst auf die zugehörige Bestätigung, bevor er mit dem Senden des nächsten Fragments fortfährt. Beim Empfang einer solchen Nachricht antwortet der Empfänger mit einer Bestätigungsnachricht, die denselben Nachrichtenzähler und (bei Bedarf) denselben Fragmentation-Header wie die Nachricht beinhaltet, für die diese Bestätigung gesendet wird.

Auf diese Weise können die Bestätigungen eindeutig der Nachricht bzw. einem

Fragment der Nachricht zugeordnet werden. Sollte der Sender keine Bestätigung für ein Fragment in einem festgelegten Zeitraum erhalten, so wird er dasselbe Fragment bis zu k mal senden. Sollte auch beim letzten Senden keine Bestätigung in einem festgelegten Zeitraum eintreffen, so werden auch die restlichen Fragmente der Nachricht nicht versendet und der lokale DPS-Stub/Skeleton mittels Callback über den Misserfolg des Sendevorgangs informiert, wo dieser anschließend behandelt wird.

Die Verwendung von Bestätigungen ist nicht für alle RPC-Aufrufe notwendig oder sinnvoll. Aus diesem Grund kann der DPS-Stub/Skeleton für jede Nachricht einzeln entscheiden, ob diese mit Bestätigungen verschickt werden soll. Bestätigungen sollten genau dann verwendet werden, wenn der entsprechende RPC-Aufruf den Zustand des Kommunikationspartners oder der Verbindung ändert. Auf die Verwendung von Bestätigungen kann jedoch verzichtet werden, wenn der Funktionsaufruf selbst über einen entsprechenden Mechanismus verfügt (z.B. Sendefunktion von TCP) oder diesen nicht benötigt (z.B. Sendefunktion von UDP).

Ein Beispiel für die Verwendung von Bestätigungen im Zusammenhang mit dem Fragmentierungsmechanismus ist als Sequenzdiagramm in Abbildung 4.6 (b) dargestellt. In diesem Beispiel sendet der Protokoll-Stub die Nachricht M_3 an den Protokoll-Skeleton. M_3 muss in insgesamt zwei Fragmenten ($F_1(M_3)$ und $F_2(M_3)$) übertragen werden. Der Client sendet $F_1(M_3)$ an den Server, der das Fragment mit einer Ack-Nachricht bestätigt, die denselben Zähler und Fragmentierungs-Header wie $F_1(M_3)$ beinhaltet. Nachdem die Bestätigung für das erste Fragment vom Client empfangen wurde, sendet der Client das zweite Fragment $F_2(M_3)$ an den Server. Die Bestätigung für das zweite Fragment geht jedoch auf dem Funkmedium verloren, weshalb der Client nach dem Ablauf des Timers T_D das Fragment erneut versendet. Diesmal wird die Bestätigung erfolgreich zum Client übertragen. Das DPS-Protokoll auf dem Server setzt M_3 aus den beiden empfangenen Fragmenten zusammen und informiert anschließend den Skeleton über den Empfang der Nachricht M_3.

4.1.5. Erkennung von Verbindungsabbrüchen

In Abhängigkeit vom Einsatzszenario und den eingesetzten DPS-Protokoll-Stubs bzw. -Skeletons kann das Erkennen von Verbindungsabbrüchen notwendig werden, da der Client auf eine aktive Verbindung zum Server angewiesen ist, um an der Kommunikation mit dem restlichen Netzwerk teilzunehmen. In geringerem Maße ist auch der Server auf eine Verbindung zum Client angewiesen, um eintreffende Nachrichten an den Client weiterleiten zu können. Hierfür bietet das DPS-Protokoll die Möglichkeit, den Status der Verbindung mittels sogenannter Heartbeat-Nachrichten zu beobachten. Dieser Mechanismus basiert auf dem Protokoll zur Knotenausfallerkennung aus dem Projekt FleGSens [107].

Heartbeat-Nachrichten beinhalten den aktuellen Nachrichtenzähler des Senders und werden von beiden Kommunikationspartnern nach einer festgelegten Zeit-

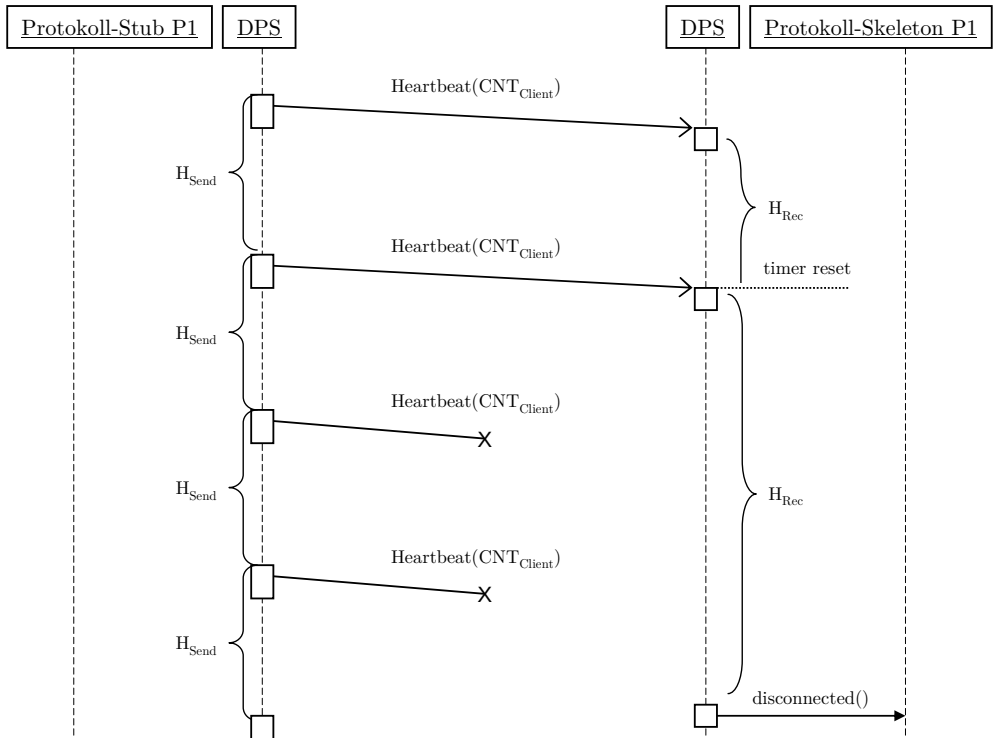


Abbildung 4.7.: Sequenzdiagramm des Mechanismus zur Erkennung von Verbindungsabbrüchen: Ein Client sendet Heartbeat-Nachrichten an seinen Server

spanne nach der letzten DPS-Nachricht verschickt (Request, Response, Ack oder vorheriges Heartbeat). Auf diese Weise wird sichergestellt, dass immer mindestens eine Nachricht in der festgelegten Zeitspanne zwischen den Kommunikationspartnern in jeder Richtung ausgetauscht wird. Gleichzeitig kann der Versand von Heartbeat-Nachrichten vermieden werden, wenn das Nachrichtenaufkommen zwischen den beiden Kommunikationspartnern ausreichend hoch ist. Beide Kommunikationspartner speichern den Zeitpunkt der letzten Empfangenen DPS-Nachricht und überprüfen regelmäßig, wie lange dieser Zeitpunkt zurückliegt. Sollte der Zeitpunkt länger als ein gegebener Schwellwert I_{rec} in der Vergangenheit liegen, so wird die Verbindung zurückgesetzt. In diesem Fall löscht der Server die Verbindung, während der Client erneut mit dem Discovery-Prozess beginnt und versucht, eine neue Verbindung aufzubauen.

Abbildung 4.7 zeigt den Mechanismus zur Erkennung von Verbindungsabbrüchen in Form eines Sequenzdiagramms. Hierbei sendet das DPS-Protokoll des Clients in regelmäßigen Abständen Heartbeat-Nachrichten an das DPS-Protokoll auf dem Server. Jedes mal, wenn das DPS-Protokoll auf dem Server eine Heartbeat-Nachricht empfängt, wird der Timer H_{Rec} neu gestartet. Die ersten beiden Heartbeat-Nachrichten erreichen den Server, alle nachfolgenden Nachrichten gehen jedoch auf dem Funkmedium verloren, weshalb der Timer H_{Rec} nicht länger neu gestartet wird. Nach dem Ablauf dieses Timers erkennt das DPS-Protokoll auf dem Server den Verbindungsabbruch und informiert den Protokoll-Skeleton.

Abbildung 4.7 zeigt lediglich den Nachrichtenaustausch in einer Richtung, weshalb die Heartbeat-Nachrichten, welche vom Server an den Client geschickt werden, nicht dargestellt sind. In Abhängigkeit von der Ursache des Nachrichtenverlust ist es jedoch möglich, dass der Ablauf in der Gegenrichtung identisch zu dem in Abbildung 4.7 dargestellten Ablauf ist, weshalb auch der Client den Verbindungsabbruch zum selben Zeitpunkt erkennen wird.

4.2. Nachrichtenformat

In diesem Abschnitt wird das Nachrichtenformat des DPS-Protokolls beschrieben. Das DPS-Protokoll arbeitet oberhalb der Sicherungsschicht von IEEE 802.15.4 und verwendet dessen Adressierungsschema, wobei sowohl 16-Bit als auch 64-Bit MAC-Adressen unterstützt werden. Die MTU (Maximum Transmission Unit) von IEEE 802.15.4 beträgt 127 Byte, was abzüglich des MAC-Headers eine maximale Paketgröße von 116 Byte (16-Bit Adressen) bzw. 104 Byte (64-Bit Adressen) für das DPS-Protokoll ermöglicht.

Das allgemeine DPS-Nachrichtenformat ist in Abbildung 4.8 dargestellt: Es besteht aus dem DPS-Feld, gefolgt von einem 4 Byte Nachrichtenzähler (Counter), der Protokoll-ID (P_{ID}), dem Payload der Nachricht und einer 4 Byte Prüfsumme. Das DPS-Feld stellt hierbei das erste Byte der DPS-Nachricht dar und beinhaltet mehrere Flags, die in Abbildung 4.9 dargestellt sind. Das DPS-Feld beginnt immer mit der Bitfolge 10, gefolgt von dem Ack-Bit, dem F-Bit und dem Nachrichtentyp. Das Ack-Bit gibt an, ob der Empfang dieser Nachricht mit einer Bestätigung vom Empfänger quittiert werden muss, während das F-Bit angibt, ob es sich um eine fragmentierte Nachricht handelt. Das Type-Feld gibt an, ob es sich bei dieser Nachricht z.B. um einen Request, eine Response, ein Ack oder eine Heartbeat-Nachricht handelt. Die Protokoll-ID dient der Adressierung eines bestimmten Protokoll-Stubs oder -Skeletons und entspricht der von der IANA vergebenen Protokollnummer, die auch für das Next-Header-Feld von IPv6 und das Protocol-Feld von IPv4 verwendet wird (z.B. 41 für IPv6 oder 14 für UDP, siehe [108]). Falls das F-Bit gesetzt ist, beinhaltet das Paket zusätzlich den Fragmentation-Header (siehe unten).

Beispiele für nicht fragmentierte DPS-Nachrichten finden sich in Abbildung 4.10 und Abbildung 4.11: Während des Verbindungsaufbaus besitzt der Payload der DPS-Nachrichten eine feste Länge von 4 Byte (siehe Abbildung 4.10), der zum Transport der Nonce und des Server-Paketzählers verwendet werden, wie in Abschnitt 4.1.2 beschrieben wurde. Heartbeat-Nachrichten hingegen beinhalten keinen Payload und haben somit eine Länge von 10 Byte (vgl. Abbildung 4.11).

Die vor dem Aufbau einer Verbindung versendeten Discovery- und Advertise-Nachrichten verwenden ebenfalls das in Abbildung 4.8 dargestellte allgemeine Nachrichtenformat, wobei der Counter der Discovery-Nachricht vom Client gewählt wird und vom Server in der Advertise-Nachricht wiederholt wird, um dem Client eine Zuordnung seiner Antwort auf die Discovery-Nachricht zu ermöglichen. Das P_{ID} Feld wird vom Client mit dem von ihm gesuchten Protokoll initialisiert.

Byte	Feld	Connect	Allow	Finish
	SRC	Client-Adresse	Server-Adresse	Client-Adresse
	DEST	Server-Adresse	Client-Adresse	Server-Adresse
1	TYPE	Connect (0)	Allow (1)	Finish (2)
4	COUNTER	CNT_{Client}		
1	P_{ID}	Protokoll-Identifikator (z.B. IPv6)		
4	PAYLOAD	N_{CONN}	CNT_{Server}	
4	CHECKSUM	Berechnet mittels K_{CONN}		

Tabelle 4.1.: Inhalt der Nachrichtenfelder bei Connect-, Allow- und Finish-Nachrichten

Der Payload einer Discovery-Nachricht kann eine Liste von Filtern beinhalten, welche durch die in Tabelle 4.2 beschriebenen 8-bit Werte repräsentiert werden und zur Bildung des Payloads hintereinander gehängt werden. Der Server bildet den Payload der Advertise-Nachricht auf dieselbe Weise.

Eine Übersicht der in den einzelnen Feldern während des Verbindungsaufbaus verwendeten Werte ist in Tabelle 4.1 zusammengefasst: Alle drei Nachrichten verwenden den vom Client gewählten Counter CNT_{Client}. Dieser Counter wird verwendet, um die Nachrichten eines Verbindungsvorgangs einander zuordnen zu können. Der Payload der Connect-Nachricht beinhaltet die Verbindungs-Nonce N_{CONN}, die zusammen mit dem paarweisen Schlüssel zwischen Client und Server zur Bildung des Verbindungsschlüssels K_{CONN} benutzt wird, der bei der Berechnung der Prüfsummen verwendet wird. Der Server antwortet mit der Allow-Nachricht auf die Verbindungsanfrage des Clients, wobei er seinen eigenen Counter CNT_{Server} im Payload mitschickt. Den Empfang des Zählers bestätigt der Client, indem er CNT_{Server} im Payload der Finish-Nachricht wiederholt.

Das Nachrichtenformat der Request- und Response-Nachrichten beinhaltet neben den bereits genannten Feldern zusätzlich noch das F_{ID}-Feld, das für die Adressierung einer Funktion innerhalb des Protokoll-Stubs und -Skeletons verwendet wird (z.B. „send()“ oder „getIPAddress()“ innerhalb der IPv6-Schicht). Falls die Länge einer DPS-Nachricht durch den Payload die MTU der Sicherungsschicht überschreitet, so kommt der vom DPS-Protokoll verwendete Fragmentierungsmechanismus zum Einsatz: Hierfür wird nach dem F_{ID}-Feld der Fragmentation-Header eingefügt, wie in Abbildung 4.12 dargestellt ist. Die Anwesenheit des Fragmentation-Header wird durch das Setzen des F-Bit im DPS-Feld signalisiert. Der Fragmentation-Header beinhaltet zwei Felder der Länge 2 Byte: Das Length-Feld, das die Länge des nicht fragmentierten Paketes in Byte angibt und das Offset-Feld, das die Position des aktuellen Fragments im nicht fragmentierten Paket angibt. Der DPS-Header wird inklusive des Fragmentation-Headers für jedes Fragment der DPS-Nachricht wiederholt und nimmt somit 11 Byte in jedem Fragment in Anspruch. Die Prüfsumme wird ebenfalls für jedes Fragment getrennt berechnet und an das entsprechende Fragment angehängt, wodurch sich eine maximale Payload-Länge von 101 Byte (89 Byte bei 64-Bit MAC-Adressen) ergibt.

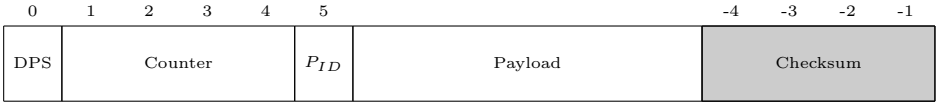


Abbildung 4.8.: Allgemeines DPS-Nachrichtenformat

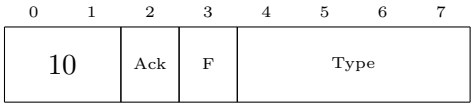


Abbildung 4.9.: Details des DPS-Feldes aus Abbildung 4.8

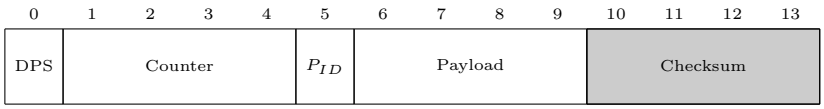


Abbildung 4.10.: Nachrichtenformat während des Aufbaus der DPS-Verbindung (Connect-, Allow- und Finish-Nachrichten)

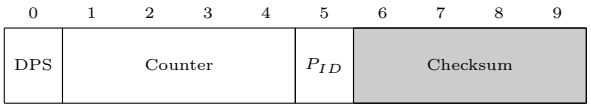


Abbildung 4.11.: Nachrichtenformat einer Heartbeat-Nachricht

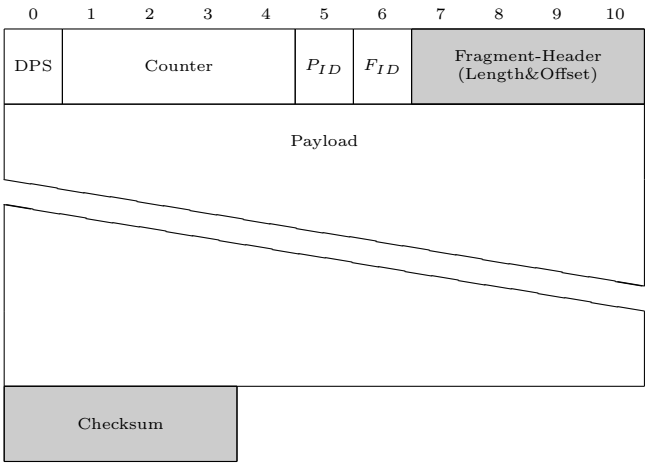


Abbildung 4.12.: Nachrichtenformat eines RPC-Aufrufs

4.2.1. Header-Kompression

Um den Overhead des DPS-Protokolls zu verringern, werden die Header-Felder der RPC-Nachrichten (Request und Response) komprimiert. Die Größe des Headers wird durch eine sparsame Kodierung der einzelnen Header-Felder verringert und zusätzlich durch die zustandsbehaftete Kompression des Nachrichtenzählers erreicht.

Dies ist in Abbildung 4.12 dargestellt: Während die Header-Felder im unkomprimierten Fall 11 Byte lang sind, können diese durch den Einsatz der Kompression auf 5 Byte reduziert werden. Die beiden Adressierungsfelder F_{ID} und P_{ID} werden hierfür in ihrer Auflösung reduziert. Statt 256 verschiedenen Protokollen mit jeweils 256 Funktionen werden nun nur noch 16 verschiedene Protokolle mit jeweils 16 Funktionen unterstützt. Zur Identifikation der einzelnen Protokolle werden hierfür die Protokoll Nummern der IANA [108] mittels einer anwendungsspezifischen Tabelle auf den Zahlenraum von 0 bis 15 abgebildet. Zusätzlich wird nur das niedrigstwertige Byte des Nachrichtenzählers in der Nachricht übertragen, während die höherwertigen Byte aus dem Verlauf der Verbindung rekonstruiert werden. Außerdem enthalten z.B. die Heartbeat-Nachrichten, die in regelmäßigen Abständen verschickt werden, den unkomprimierten Nachrichtenzähler.

Auch der Fragmentation-Header wurde in seiner Größe reduziert: Das Length-Feld besitzt nun eine Länge von 11 Bit, wodurch eine maximale Paketlänge von 2^{11} Byte = 2048 Byte ermöglicht wird. Das Offset-Feld gibt nun den Index des Fragments an (maximal 32 Fragmente / Paket) und nicht wie vorher die Position des Fragments im nicht fragmentierten Paket. Neben der zustandslosen Kompression dieser Felder wird zusätzlich noch die zustandsbehaftete Kompression des Nachrichtenzählers verwendet: Jede komprimierte DPS-Nachricht beinhaltet lediglich das niedrigstwertige Byte des 4 Byte langen Nachrichtenzählers. Die übrigen Byte werden auf der Empfängerseite aus dem Nachrichtenverlauf rekonstruiert.

Dies ist möglich, da die Prüfsumme vom Sender vor der Kompression des Header berechnet wird, wodurch der vollständige Nachrichtenzähler in der Prüfsumme enthalten ist. Der Empfänger dekomprimiert also alle Felder beim Empfang der Nachricht und rekonstruiert den Nachrichtenzähler, indem er die drei höchstwertigen Byte des vorherigen Nachrichtenzählers mit dem komprimierten Zähler verknüpft. Stimmt die Prüfsumme nicht überein, so erhöht der Empfänger den rekonstruierten Zähler in 256er Schritten und berechnet die Prüfsumme erneut. Stimmt die Prüfsumme nach k-maligem Erhöhen (k=2) nicht überein, wird die Nachricht verworfen.

Damit die Rekonstruktion des Nachrichtenzählers funktioniert, muss der Zähler in regelmäßigen Abständen synchronisiert werden. Hierfür werden die in regelmäßigen Abständen zwischen beiden Kommunikationspartnern ausgetauschten Heartbeat-Nachrichten verwendet, die immer unkomprimiert versendet werden. Das Intervall, in dem Heartbeat-Nachrichten verschickt werden bzw. nach dem ein Verbindungsabbruch erkannt wird (siehe Abschnitt 4.1.5) sollte deshalb

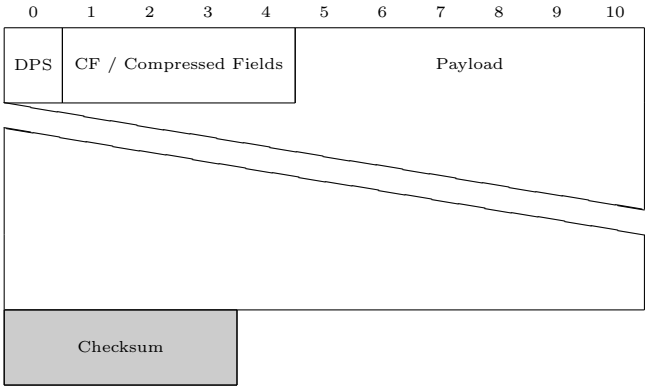


Abbildung 4.13.: Nachrichtenformat eines RPC-Aufrufs unter Verwendung der Header-Kompression

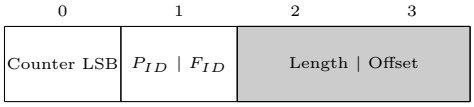


Abbildung 4.14.: Details des CF-Feldes aus Abbildung 4.13

so gewählt werden, dass nicht mehr als $k * 256$ Nachrichten hintereinander verloren gehen. Falls das DPS-Protokoll ohne Heartbeats eingesetzt wird und mit einem hohen Nachrichtenverlust zu rechnen ist, so kann der Sensornetzbetreiber die Verwendung des Kompressionsverfahrens deaktivieren.

4.3. Sicherheit

Die in diesem Abschnitt vorgestellten Mechanismen sollen dem Schutz des DPS-Protokolls vor einem Angreifer dienen. Hierfür wird zunächst dieser abstrakte Angreifer in Form eines Angreifermodells in Abschnitt 4.3.1 konkretisiert und die Fähigkeiten und Ziele des Angreifers beschrieben. Anschließend werden die möglichen Angriffsarten in Abschnitt 4.3.2 definiert, gefolgt von den eingesetzten Sicherheitsmechanismen.

4.3.1. Angreifermodell

Ein Angreifer ist ein aktiver und feindseliger Teilnehmer des Netzes, der durch die ihm zur Verfügung stehenden Mittel versucht, den Ablauf der im Netz stattfinden Kommunikation zu stören oder zu manipulieren. In Abhängigkeit von dem gewählten Angreifermodell stehen dem Angreifer nur bestimmte Mittel zur Verfügung, die er einsetzen kann. Ein häufig verwendetes Angreifermodell ist hierbei der Dolev-Yao-Angreifer [109], das von Danny Dolev und Andrew Yao entwickelt wurde und das bereits im Rahmen des Projekts FleGSens vom Autor dieser Arbeit verwendet wurde [110]. Der Dolev-Yao-Angreifer verfügt über die folgenden Fähigkeiten: der Angreifer...

- kann jede Nachricht empfangen, die im Netz versendet wird.
- kann Nachrichten an jeden anderen Teilnehmer des Netzes senden.
- kann den Empfang einer beliebigen Nachricht im Netzwerk verhindern.
- kann eine oder mehrere beliebige Identitäten einnehmen.
- kennt alle im Netz verwendeten Protokolle und Nachrichtenformate.

Der Angreifer verfügt jedoch auch über einige Einschränkungen. So kann der Dolev-Yao-Angreifer keine Verschlüsselungsverfahren umgehen, d.h. er kann keine verschlüsselten Daten entschlüsseln und keine Daten verschlüsseln, solange er den Schlüssel nicht kennt. Er kann den Schlüssel jedoch speichern, falls dieser unverschlüsselt im Netzwerk verschickt wird (z.B. durch ein Schlüsselverteilungsprotokoll). Darüber hinaus hat der Angreifer keinen direkten Zugang zu der Hardwareplattform, d.h. er kann den Speicher des Knoten nicht zur Laufzeit auslesen, um z.B. an das verwendete Schlüsselmaterial zu gelangen.

Zusammenfassend kann man den Angreifer als Man-in-the-Middle betrachten, der sämtliche im Netz versendeten Nachrichten empfangen und manipulieren kann, indem er die Nachricht verändert, durch eine eigene ersetzt oder löscht. Im nachfolgenden Abschnitt 4.3.2 sollen zunächst einige Angriffsarten vorgestellt werden, die von einem Dolev-Yao-Angreifer gegen das DPS-Protokoll eingesetzt werden können.

Zu den Angriffen, die von einem Dolev-Yao-Angreifer durchgeführt werden können, zählt unter anderem der sogenannte Replay-Angriff. Bei diesem Angriff speichert der Angreifer eine im Netz versendete Nachricht, um diese zu einem späteren Zeitpunkt an einen oder mehrere Teilnehmer des Netzes (erneut) auszuliefern. Wird das Ausliefern der Nachricht an einem Ort durchgeführt, der außerhalb der Funkreichweite des ursprünglichen Senders liegt, spricht man in diesem Fall von einem Wormhole-Angriff. Eine andere Angriffsart stellt das Spoofing dar, bei dem der Angreifer die Absenderadresse einer Nachricht verändert, um in die Rolle eines anderen Netzteilnehmers zu schlüpfen.

Eine Möglichkeit für einen solchen Replay-Angriff wäre es, eine bestimmte Anzahl von aufeinander folgenden Nachrichten des Senders zu speichern und den Empfang dieser Nachrichten durch den Empfänger für eine gewählte Zeitspanne zu verzögern (Slow-Motion-Replay). In diesem Fall verhindert der Angreifer also den Empfang der Nachrichten durch den Empfänger. Diese Zeitspanne wird durch den Einsatz des Mechanismus zur Erkennung von Verbindungsabbrüchen nach oben begrenzt. Diese Angriffsart kann also nicht dazu verwendet werden, die Auslieferung von Nachrichten für einen beliebig langen Zeitpunkt zu verhindern. Der Angriff kann jedoch dafür verwendet werden, die Latenz und den Datendurchsatz der Verbindung zwischen den beiden Kommunikationspartnern zu verschlechtern.

4.3.2. Angriffe

In diesem Abschnitt werden die Angriffe beschrieben, die für das DPS-Protokoll relevant sein können. Ein Angreifer wird nicht notwendigerweise auf alle Angriffe zurückgreifen, sondern sich für einen spezifischen Angriff oder eine Kombination von Angriffen entscheiden. Das Interesse des Angreifers wird in erster Linie durch das Anwendungsszenario bestimmt, in dem das DPS-Protokoll eingesetzt wird, weshalb die hier erwähnten Angriffe nur allgemeine Angriffe darstellen.

In Abhängigkeit von der gewählten Anwendung müssen die Anwendungsprotokolle unabhängig vom DPS-Protokoll abgesichert werden, so wie dies im Rahmen des FleGSens Projektes geschehen ist (siehe Abschnitt 2.1.3). Ein Beispiel hierfür ist die Absicherung eines Lokalisierungsprotokolls gegen Wormhole-Angriffe [64], was nicht vom DPS-Protokoll geleistet werden kann, da dies zusätzliches anwendungsspezifisches Wissen voraussetzt.

A1: Veränderung des Nachrichteninhalts

Der Inhalt einer DPS-Nachricht beinhaltet die serialisierten Daten der Stubs und Skeletons und dient somit dem Transport von protokollspezifischen Informationen oder der Weiterleitung von Nachrichten der entsprechenden Protokollschicht. Ein Angreifer hat folglich ein Interesse daran, den Inhalt dieser Nachrichten zu verändern, um z.B. die transportierten Sensordaten zu manipulieren. Der Angreifer wird hierfür einzelne Bytes der Nachricht verändern, Bytes hinzufügen oder entfernen.

A2: Vortäuschung einer falschen Identität

Um die Kommunikation zu stören, wird der Angreifer versuchen, das Nachrichtenaufkommen im Netz zu erhöhen, zusätzliche DPS-Verbindungen zu erstellen oder existierende DPS-Verbindungen zu schließen. Da der Angreifer die verwendeten Protokolle und Nachrichtenformate kennt, kann er hierfür entweder selbst Nachrichten erzeugen oder aber mitgehörte Nachrichten unter Veränderung der Absende- und/oder Zieladresse verschicken.

A3: Erhöhung des Nachrichtenverlusts

Der Angreifer kann die Auslieferung beliebiger Nachrichten im Netz verhindern und wird auf diese Weise versuchen, den Nachrichtenverlust zu erhöhen, um die Menge der empfangenen Daten zu reduzieren. Dies kann entweder dazu dienen, die Dienstgüte der Anwendung zu reduzieren (z.B. durch den Verlust von Sensordaten) oder bestimmte Knoten vom Rest des Netzes zu trennen.

A4: Verhinderung/Verzögerung der Erkennung von Verbindungsabbrüchen

Die Erkennung von Verbindungsabbrüchen wird von einem DPS-Client benötigt, um die Teilnahme am Netzwerk sicherstellen zu können, nachdem die Verbindung zu dem entsprechenden Server nicht länger verfügbar ist. Ein Angreifer wird deshalb versuchen, die Erkennung dieses Verbindungsabbruchs zu verzögern oder ganz zu verhindern. Ein Verbindungsabbruch kann folgende Gründe haben:

- Veränderung der Netzwerktopologie bzw. des Links: Sowohl der Client als auch der Server sind weiterhin aktiv, können jedoch nicht mehr direkt miteinander kommunizieren.
- Ausfall eines Knotens: Entweder der Client oder der Server fallen aus - z.B. durch Verlust der Energieversorgung oder Absturz des Programms.

4.3.3. Sicherheitsmechanismen

Basierend auf den im vorangegangenen Abschnitt definierten Angriffen werden in diesem Abschnitt die Sicherheitsmechanismen vorgestellt. Diese werden vom DPS-Protokoll verwendet, um die Angriffe zu erkennen oder zu verhindern. Hierbei werden insgesamt zwei Mechanismen verwendet: Ein in jeder Nachricht enthaltener Nachrichtenzähler und ein Message-Authentication-Code (MAC), der an die Nachricht angehängt wird. Diese beiden Mechanismen und ihre Funktionsweise werden im Folgenden genauer erläutert.

Nicht in jedem Anwendungsszenario ist der Einsatz von Sicherheitsmechanismen erforderlich oder möglich. Das DPS-Protokoll wurde deshalb so konzipiert, dass die hier beschriebenen Sicherheitsmechanismen optional zum Schutz des Protokolls eingesetzt werden können. Der Anwendungsentwickler kann auf diese Sicherheitsmechanismen verzichten, wenn dies in seinem Anwendungsszenario möglich ist: Entweder weil die Existenz eines Angreifers ausgeschlossen werden kann oder aber weil andere Mechanismen einen ausreichenden Schutz bieten (z.B. auf der Sicherungsschicht). Darüber hinaus ist der Einsatz dieser Sicherheitsmechanismen mit zusätzlichem Overhead verbunden, da die Implementierung der Mechanismen sowohl Programmspeicher als auch dynamischen Speicher und Rechenkapazität zur Laufzeit benötigt. Dies reduziert den Speicher, der für die Anwendung und die restlichen Protokolle zur Verfügung steht, verringert den Datendurchsatz und erhöht die Latenz. Diese Faktoren müssen vom Anwendungsentwickler gegeneinander abgewogen werden.

Bei der Betrachtung der zwei nachfolgenden Sicherheitsmechanismen werden die Eigenschaften des verwendeten Dolev-Yao-Angreifers berücksichtigt, d.h. der Angreifer besitzt nicht die Möglichkeit, einen MAC selbst zu berechnen, ohne den zu seiner Berechnung benötigten Schlüssel zu kennen. Durch die Wahl eines hierfür geeigneten MAC-Berechnungsverfahrens (z.B. AES-CBC-MAC) wird es ihm darüber hinaus erschwert, den Klartext gezielt zu verändern, so dass der MAC auch für die veränderte Nachricht gültig ist.

Message-Authentication-Code

Jede Nachricht, die vom DPS-Protokoll verschickt wird, beinhaltet eine Prüfsumme, welche die Integrität und die Authentizität der Nachricht schützt. Die Prüfsumme wird über den gesamten Nachrichteninhalt inklusive der Quell- und Zieladresse in Form eines Message-Authentication-Code (MAC) berechnet. Ein MAC unterscheidet sich von einer Prüfsumme, die mittels eines Hashfunktion berechnet wurde (wie z.B. MD5 oder SHA-1), da er zusätzlich zu den zu schützenden Daten einen geheimen Schlüssel als Eingabe erhält, der in die Berechnung einfließt. Ohne die Kenntnis dieses geheimen Schlüssels ist es einem Angreifer nicht möglich, einen gültigen MAC für eine gegebene Nachricht zu berechnen. Hierdurch wird sowohl die Integrität als auch die Authentizität der Nachricht sichergestellt, indem der Empfänger beim Empfang der Nachricht ebenfalls den MAC für diese Nachricht berechnet. Stimmen der empfangene und der berechnete MAC überein, so wurde der Nachrichteninhalt mit hoher Wahrscheinlichkeit nicht verändert und die Nachricht wurde von dem angegebenen Sender gesendet.

Der für die Berechnung des MAC verwendete paarweise Schlüssel muss vor Beginn des DPS-Protokolls zwischen dem Sender und dem Empfänger ausgehandelt werden, wofür ein für das Anwendungsszenario geeignetes Schlüsselverteilungsprotokoll eingesetzt werden muss (z.B. vorverteilte Schlüssel, das Key-Infection-Protokoll [111] oder HARPS [112]). Für die Berechnung des MAC während des Discovery-Prozesses wird ein globaler oder Gruppenschlüssel verwendet, da in dieser Protokollphase Broadcast-Kommunikation zum Einsatz kommt und somit kein paarweiser Schlüssel zwischen Sender und Empfänger zur Verfügung steht. Für die restlichen Protokollphasen wird dann der paarweise Schlüssel zwischen Client und Server verwendet, der mit der beim Verbindungsaufbau ausgehandelten NONCE zu einem Sitzungsschlüssel kombiniert wird.

Nachrichtenzähler

Neben der Prüfsumme beinhaltet jede Nachricht, die vom DPS-Protokoll verschickt wird, einen Nachrichtenzähler. Dieser Nachrichtenzähler wird von beiden Kommunikationspartnern zu Beginn der Verbindung zufällig gewählt und anschließend bei jedem Sendevorgang erhöht. Auf diese Weise unterscheiden sich zwei beliebige Nachrichten, auch wenn sie denselben Payload besitzen, immer mindestens durch den Nachrichtenzähler. Der Empfänger speichert jeweils den letzten empfangenen Nachrichtenzähler des Senders und akzeptiert anschließend nur solche Nachrichten, die einen höheren Nachrichtenzähler beinhalten (und deren MAC gültig ist). Da der Zahlenbereich des verwendeten Nachrichtenzählers endlich ist und sich deshalb nach endlicher Zeit alle Nachrichtenzähler wiederholen, müssen beide Kommunikationspartner spätestens zu diesem Zeitpunkt eine neue Verbindung aufbauen um eine neue Nonce auszuhandeln, wodurch sich die MACs von zwei Nachrichten auch dann unterscheiden, wenn sie denselben Payload und denselben Nachrichtenzähler besitzen.

Auf diese Weise wird es dem Angreifer erschwert, einen Replay-Angriff durchzuführen, er kann diesen jedoch in Form eines Slow-Motion-Replay-Angriffs durchführen und somit die Kommunikationsgeschwindigkeit um einen nach oben begrenzten Faktor verringern, um z.B. die Erkennung von Verbindungsabbrüchen zu verzögern.

Hinweis: Keine Verschlüsselung

Das DPS-Protokoll schützt ausschließlich die Integrität und Authentizität der Nachrichten, während die Vertraulichkeit der versendeten Daten nicht sichergestellt wird. Integrität und Authentizität spielen in einem Sensornetz eine größere Rolle als Vertraulichkeit, wie die Autoren von TinySec in [113] näher erläutern: Eine Veränderung des Nachrichteninhalts muss immer erkannt werden, da ein Angreifer sonst auf einfache Art und Weise auch bei verschlüsselten Nachrichten durch einen Replay-Angriff und das Kippen einzelner Bits fehlerhafte Nachrichten in das Netzwerk einschleusen könnte. Vertraulichkeit sollte in einem IP-Netzwerk entweder auf der Sicherungsschicht (vgl. WPA2 bei WLAN), Vermittlungsschicht (vgl. IPsec) oder aber auf der Transportschicht (Ende-zu-Ende Sicherheit, vgl. TLS/SSL) realisiert werden.

Falls die Vertraulichkeit der versendeten Daten zu den Schutzzielen der gewählten Sensornetzanwendung gehört, so empfiehlt der Autor die Verwendung der hierfür vorgesehenen Mechanismen von IEEE 802.15.4 zur Absicherung der Sicherungsschicht, z.B. mittels eines netzwerkweiten Schlüssels. Falls das verwendete Angreifermodell jedoch annimmt, dass der Angreifer diesen Schlüssel kennt (z.B. durch das Auslesen des Speichers der Sensorknoten) oder diese Mechanismen auf andere Art umgehen kann, so sollten andere Schutzmechanismen eingesetzt werden.

4.3.4. Erweiterungen

In den vorangegangenen Abschnitten wurden die einzelnen Protokollphasen und -mechanismen des DPS-Protokolls inklusive der Sicherheitsmechanismen vorgestellt. In diesem Abschnitt werden einige zusätzliche Erweiterungen vorgestellt, die kein fester Bestandteil des Protokolls sind. Diese Erweiterungen sollen entweder die Leistung des Protokolls verbessern oder dessen Einsatzzweck vergrößern.

Nachrichtenkompression

Die Kompression des Nachrichteninhalts kann mittels eines verlustfreien Kompressionsalgorithmus durchgeführt werden, wie z.B. LZW [114] (Lempel-Ziv-Welch). In Abhängigkeit von dem zu erwartenden Kompressionsgrad der Anwendungsdaten kann dies die Latenz und den Datendurchsatz des DPS-Protokolls verbessert. Hierbei ist jedoch zu beachten, dass Anwendungsdaten sich häufig

nur stark eingeschränkt oder überhaupt nicht komprimieren lassen und dass die Anwendung des Kompressionsalgorithmus unabhängig vom Kompressionsgrad Zeit und Speicherplatz benötigt. Ob der Einsatz eines solchen Kompressionsverfahrens sinnvoll ist, hängt von den auftretenden Anwendungsdaten ab und sollte vom Anwendungsentwickler sorgfältig untersucht werden.

Einsatz in heterogenen Netzen

Durch die Implementierung des DPS-Protokolls auf unterschiedlichen Sensornetzplattformen können Protokolle und Dienste plattformübergreifend genutzt werden. In heterogenen Netzen, die unterschiedliche Hard- und Softwareplattformen einsetzen, ist es auf diese Weise möglich, die Implementierungen der einzelnen Plattformen flexibel miteinander zu verbinden. Bei diesem Einsatzzweck kann das DPS-Protokoll ähnlich einem klassischen RPC-Framework eingesetzt werden.

Nutzung der Testbed-Infrastruktur

Bei der Verwendung eines Testbeds können Protokollschichten zusätzlich zu der Verteilung im Sensornetz auch über die Infrastruktur des Testbeds auf einen angeschlossenen PC ausgelagert werden. Auf diese Weise können Protokolle genutzt werden, die noch nicht für die Sensornetzplattform implementiert wurden oder es können die vorhandenen Protokolle mit einem zentralisierten (optimalen) Algorithmus verglichen werden. Beispiele hierfür sind unter anderem ein Schlüsselverteilungsprotokoll, das paarweise Schlüssel zwischen allen Sensorknoten im Netzwerk zur Verfügung stellt oder ein Routing-Protokoll, das den kürzesten Pfad zwischen zwei Sensorknoten im Netzwerk berechnen kann. Beide Algorithmen können sowohl verteilt auf der Sensorknotenplattform implementiert werden als auch zentralisiert auf einem PC, der über die Infrastruktur des Testbeds mit den Sensorknoten kommuniziert. Beide Ansätze werden im Rahmen der Evaluation verwendet, wobei vor allem die Nutzung der Testbed-Infrastruktur zur Umsetzung eines zentralisierten Routing-Protokolls in Abschnitt 5.7 vorgestellt wird.

4.3.5. Anforderungen und Beschränkungen

Der Einsatz des DPS-Protokolls ist mit einigen Anforderungen verbunden. Dies betrifft vor allem die Topologie des Sensornetzes und die Knotenplatzierung. Diese Anforderung ergibt sich aus dem Client-/Server-Prinzip des DPS-Protokolls, bei dem ein Client auf eine Verbindung zu einem Server angewiesen ist, um mit dem Rest des Netzwerks kommunizieren zu können. Dies muss bereits bei der Planung und bei der Ausbringung des Sensornetzes bedacht werden: Falls die Position der einzelnen Knoten genau geplant und kontrolliert werden kann, so muss dafür Sorge getragen werden, dass jeder Client in Funkreichweite zu

mindestens einem Server ausgebracht wird. Darüber hinaus muss in Abhängigkeit von den Anforderungen der Anwendung darauf geachtet werden, dass alle Server ein zusammenhängendes Netz bilden oder, falls die Anwendung dies zulässt, mehrere zusammenhängende Netze. Ist die kontrollierte Ausbringung der einzelnen Sensorknoten nicht möglich oder muss die Ausbringung komplett zufällig erfolgen, so muss durch ein entsprechendes Verhältnis von Servern zu Clients dafür gesorgt werden, dass die Anforderungen der Anwendung eingehalten werden können. Die genaue Planung und Ausbringung eines Sensornetzes stellt eine komplexe und schwierige Aufgabe dar, die bereits in vielen Publikationen und für viele unterschiedliche Anwendungsszenarien beschrieben wurde. Eine genauere Untersuchung dieser Anforderung für das DPS-Protokoll findet sich in Abschnitt 6.1.

Eine weitere Anforderung ergibt sich aus den im Abschnitt 4.3.3 vorgestellten Sicherheitsmechanismen. Der Einsatz der Sicherheitsmechanismen setzt das Vorhandensein eines Schlüsselverteilungsprotokolls voraus, das in der Lage ist, paarweise Schlüssel zwischen allen Knoten im Netzwerk zur Verfügung zu stellen. Des Weiteren muss ein Algorithmus zur Berechnung des MAC auf der Sensorknotenplattform zur Verfügung stehen, der entweder in Software implementiert sein kann oder aber Hardwarefunktionen verwenden kann.

5. Implementierung

Dieses Kapitel beschreibt die Implementierung des DPS-Protokolls auf der Grundlage des im vorangegangenen Kapitel beschriebenen Entwurfs. Zunächst wird der Protokollablauf in Abschnitt 5.1 beschrieben, wobei die Implementierung des Discovery-Prozesses und des Verbindungsaufbaus in Abschnitt 5.1.1 vorgestellt wird. Als nächstes wird in Abschnitt 5.1.2 die Implementierung des Nachrichtenaustauschs und der Erkennung von Nachrichtenverlust beschrieben, die auf den vom DPS-Protokoll verwendeten Nachrichtenzählern basiert. Für die Kommunikation mit den Protokoll-Stubs und -Skeletons werden hierbei die in Abschnitt 5.1.3 beschriebenen Event-Handler sowie die in Abschnitt 5.1.4 beschriebenen Nachrichtenqueues verwendet. Die Implementierung des vom DPS-Protokoll verwendeten Fragmentierungsmechanismus wird im darauf folgenden Abschnitt 5.1.5 beschrieben, gefolgt von der Implementierung des Mechanismus zur Erkennung von Verbindungsabbrüchen mittels Heartbeat-Nachrichten.

Anschließend wird in Abschnitt 5.2 die Berechnung des Message-Authentication-Code mittels des AES-CCM-MAC-Algorithmus beschrieben. Im darauf folgenden Abschnitt 5.2.1 wird vorgestellt, wie die vom DPS-Protokoll verwendeten Schlüssel für den Aufbau und den Betrieb der DPS-Verbindungen berechnet werden, gefolgt von den für die Verbindungs-Nonce und -Counter verwendeten Pseudo-Zufallszahlengenerator in Abschnitt 5.3 sowie die Nachrichtenkompression in Abschnitt 5.4.

Um die Verwendung des DPS-Protokolls zu veranschaulichen, wird die Implementierung von zwei Beispielen für die vom DPS-Protokoll verwendeten Protokoll-Stubs und -Skeletons in Abschnitt 5.5 beschrieben. Hierbei wird zunächst die konzeptionelle Vorgehensweise bei der Auswahl der Protokollschicht vorgestellt, gefolgt von der Beschreibung der Implementierung des Stubs und Skeletons des IPv6-Protokolls in Abschnitt 5.6.

Anschließend wird vorgestellt, wie das DPS-Protokoll zusätzlich zur Teilung von Protokoll-Implementierungen zwischen benachbarten Sensorknoten auch zur Integration von zentralisierten Protokollen verwendet werden kann. Hierfür dient ein Routing-Protokoll, das in Abschnitt 5.7 beschrieben wird und das vom Sensornetz über die Infrastruktur des WISEBED-Testbeds verwendet werden kann, um Routen im Sensornetz zu berechnen.

Den Abschluss des Kapitels bildet eine Beschreibung der Implementierung des DPS Protokolls für die Wiselib in Abschnitt 5.8.

5.1. Protokollablauf

In diesem Abschnitt findet eine Beschreibung des Protokollablaufs statt. Zu Beginn wird hierbei die Implementierung des Discovery-Prozesses und des Verbindungsaufbaus erläutert, gefolgt von der Implementierung des Nachrichtenaustauschs und zur Erkennung von Nachrichtenverlust in Abschnitt 5.1.2. Für die Kommunikation mit den vom DPS-Protokoll verwendeten Protokoll-Stubs und -Skeletons werden hierbei die in Abschnitt 5.1.3 beschriebenen Event-Handler sowie die in Abschnitt 5.1.4 beschriebenen Nachrichtenqueues verwendet. Den Abschluss dieses Abschnitts bildet die Implementierung des Fragmentierungsmechanismus in Abschnitt 5.1.5, gefolgt vom Mechanismus zur Erkennung von Verbindungsabbrüchen mittels Heartbeat-Nachrichten in Abschnitt 5.1.6.

Eine Beschreibung dieser Abläufe wurde erstmals auf der CPScom 2012 in Besançon [104] sowie auf der CPScom 2013 in Peking [105] veröffentlicht.

5.1.1. Discovery-Prozess und Verbindungsaufbau

Bevor der Client eine DPS-Verbindung zu einem Server aufbauen kann, verwendet er den in Abschnitt 4.1.1 beschriebenen Discovery- und Advertisement-Mechanismus, um alle Server in seiner Nähe zu finden, die das von Ihm gesuchte Protokoll implementieren. Hierbei folgen Client und Server dem in Abbildung 5.1 dargestellten Ablauf: Der Client initialisiert den Zähler k auf den Wert Null, der die Anzahl der versendeten Discovery-Nachrichten zählt. Zusätzlich wird eine Liste zur Speicherung der erwarteten Advertise-Nachrichten angelegt. Anschließend beginnt der Client, mindestens K_D Discovery-Nachrichten im Abstand von T_D ms zu senden, bis er mindestens eine Advertise-Nachricht empfangen hat. Die Ermittlung dieser beiden Parameter wird im Folgenden beschrieben.

Der Parameter T_D wurde auf den Wert 30 ms festgelegt, da für das Senden der Nachricht, die Verarbeitung und den Empfang der Antwort mindestens 19 ms benötigt werden (vgl. Parameter f_0 für „DPS“ in Tabelle 6.1 auf Seite 122). Dieser Wert erhöht sich in Abhängigkeit von der Länge der Nachricht um maximal 8 ms (vgl. Parameter f_s in Tabelle 6.1), woraus sich eine Gesamtdauer von 27 ms ergibt, welche auf 30 ms aufgerundet wurde.

Da Nachrichten auf dem Funkmedium verloren gehen können, wird die Discovery-Nachricht mehrmals gesendet. Hierdurch wird die Wahrscheinlichkeit erhöht, dass ein benachbarter Server diese Nachricht empfangen kann. Um einen Anhaltspunkt für die Wahl der richtigen Sendehäufigkeit zu erhalten, wird im Folgenden eine theoretische Analyse der kumulativen Verlustwahrscheinlichkeit angestellt. Hierfür wird die durchschnittliche Paketankunftsrate des Funkmediums als P_{rec} definiert, d.h. eine Nachricht geht mit einer Wahrscheinlichkeit von $P_{loss} = 1 - P_{rec}$ bei einmaligem Senden verloren. Wird eine Nachricht mehrmals gesendet, geht sie bei jedem Sendevorgang mit dieser Wahrscheinlichkeit verloren. Gesucht ist nun die Wahrscheinlichkeit, mit der mindestens eine Nachricht bei k-maligem Senden ankommt. Diese kumulative Wahrscheinlichkeit kann mittels

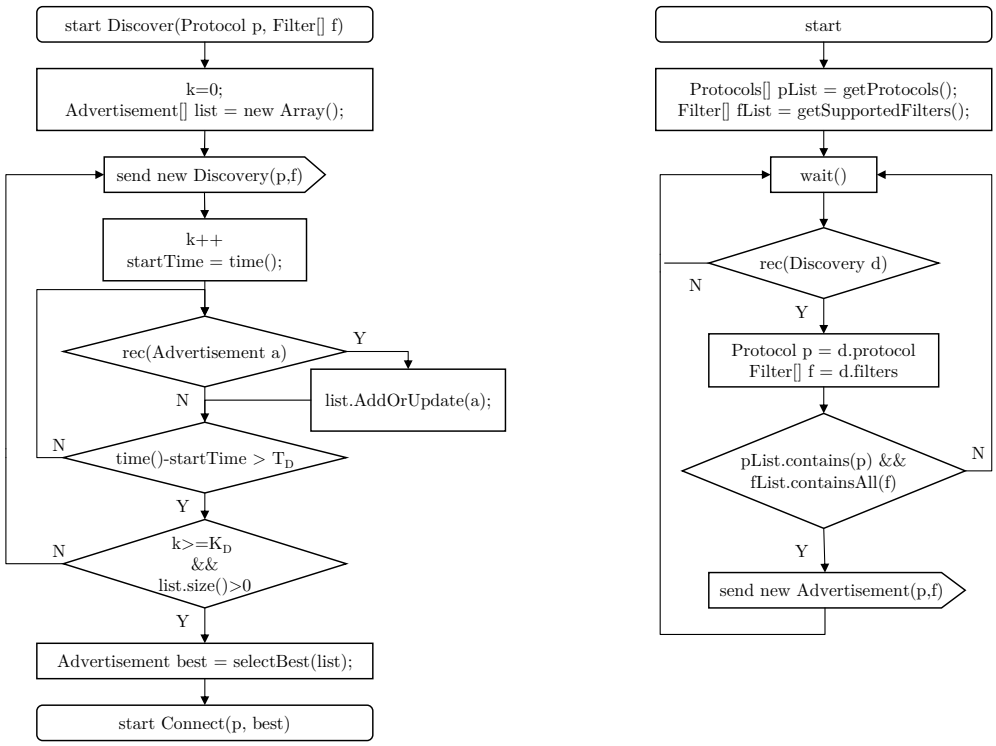


Abbildung 5.1.: Ablauf des Discovery- und Advertisement-Mechanismus auf Client und Server

der folgenden Formel berechnet werden:

$$P_{cumulative\_loss}(P_{rec}, k) = (1 - P_{rec})^k \tag{5.1}$$

Bei einer Paketankunftsrate von 90 % beträgt somit die Wahrscheinlichkeit, dass eine Nachricht bei einmaligem Senden nicht ankommt $P_{cumulative\_loss}(0{,}9,\ 1) = (1 - 0{,}9) = 10\,\%$. Die Wahrscheinlichkeit bei zweimaligem Senden hingegen beträgt $P_{cumulative\_loss}(0{,}9,\ 2) = 10\,\% * 10\,\% = 1\,\%$. Die resultierenden Werte für Paketankunftsraten zwischen 50 % und 99 % und für $k = \{1, 2, 3, 4, 5\}$ sind in Abbildung 5.2 dargestellt. Diese Werte spiegeln die Wahrscheinlichkeit wieder, dass eine Nachricht bei k-maligem Senden bei einem Empfänger ankommt.

Da jedoch auch die Antwort des Empfängers auf dem Funkmedium verloren gehen kann, ist die Wahrscheinlichkeit von Interesse, dass sowohl die Nachricht als auch deren Antwort nicht auf dem Funkmedium verloren geht. Dies entspricht genau der Wahrscheinlichkeit $P_{symm\_cumulative\_loss}(P_{rec}, k)$, da die Antwort mit derselben Wahrscheinlichkeit verloren gehen kann:

$$P_{symm\_cumulative\_loss}(P_{rec}, k) = (1 - (P_{rec})^2)^k \tag{5.2}$$

Die resultierenden Werte für Paketankunftsraten zwischen 50 % und 99 % und für $k = \{1, 2, 3, 4, 5\}$ sind in Abbildung 5.3 dargestellt. Soll z.B. bei einer

durchschnittlichen Paketankunftsrate von 80 % und k-maligem Senden eine Wahrscheinlichkeit von $P_{symm\_cumulative\_loss} < 5\,\%$ erzielt werden, so muss $k = 3$ gewählt werden. Auf diese Weise wird sichergestellt, dass bei drei gesendeten Discovery-Nachrichten mindestens eine Advertise-Nachricht mit einer Wahrscheinlichkeit von >95 % von einem ausgewählten Server empfangen werden kann. Aus diesem Grund wird $K_D = 3$ gewählt. Diese Untersuchung betrachtet die Server in Funkreichweite des Clients einzeln, d.h. für jeden der Server besteht eine Wahrscheinlichkeit von >95 %, dass die zugehörige Advertise-Nachricht empfangen wird.

Nachdem eine Advertise-Nachricht empfangen wurde, wird diese in der Advertise-Liste gespeichert, wobei existierende Nachrichten, die vom gleichen Server empfangen wurden, mittels der Informationen aus der neuen Nachricht aktualisiert werden. Die Liste speichert folglich einen Eintrag für jeden Server, von dem eine Advertise-Nachricht empfangen wurde, wobei die Anzahl der maximal in der Liste enthaltenen Einträge nur durch den zur Verfügung stehenden Speicher begrenzt ist.

Die Liste beinhaltet hierbei neben der eigentlichen Nachricht noch zusätzliche Metainformationen wie z.B. die Signalqualität, Signalstärke und die Anzahl der von diesem Server empfangenen Nachrichten. Empfängt ein Client beispielsweise eine Advertise-Nachricht von Server A und drei Nachrichten von Server B, so beinhaltet die Liste anschließend zwei Einträge: Die Advertise-Nachricht von Server A (mit Zähler eins) sowie die letzte Advertise-Nachricht von Server B (mit Zähler drei) zusammen mit der niedrigsten jeweils gemessenen Signalqualität und Signalstärke zu dem entsprechenden Server. Anschließend wählt der Client die Beste in der Liste gespeicherte Advertise-Nachricht aus, wobei hierbei als Metrik entweder die Signalstärke, Signalqualität oder die Anzahl der empfangenen Nachrichten (oder eine Kombination) verwendet werden kann.

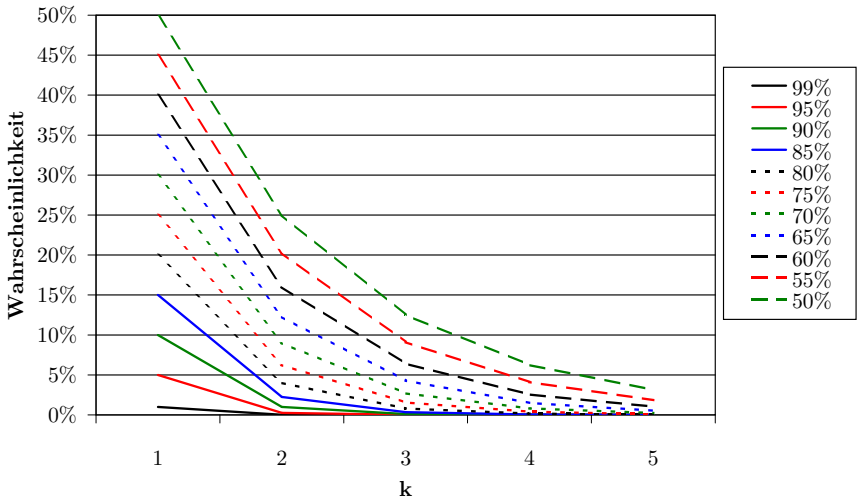


Abbildung 5.2.: Wahrscheinlichkeit, dass eine Nachricht bei k-maligem Senden nicht beim Empfänger ankommt (in Abhängigkeit von der Paketankunftsrate)

Auf der Gegenseite initialisiert der Server nach dem Start des DPS-Protokolls zunächst eine Liste der von ihm implementierten Protokolle sowie der von ihm unterstützten Filter. Empfängt ein Server nun eine Discovery-Nachricht von einem Client, so überprüft er, ob er das angefragte Protokoll und alle gewünschten Filter implementiert. Nur wenn beide Kriterien zutreffen, antwortet der Server anschließend mit einer Advertise-Nachricht. Nachdem der Client aus seiner Liste der gespeicherten Advertise-Nachrichten den besten Server ausgewählt hat, beginnt der Verbindungsaufbau mittels des Drei-Wege-Handschlags. Dieser wird vom Client nach dem in Abschnitt 4.1.2 beschrieben Ablauf initiiert.

5.1.2. Nachrichtenaustausch und Erkennung von Nachrichtenverlust

Nachdem die Verbindung aufgebaut wurde, können Client und Server RPC-Nachrichten miteinander austauschen. Wie in Abschnitt 4.2 beschrieben, beinhalten sämtliche RPC-Nachrichten einen Nachrichtenzähler und einen CCM-MAC (Counter und Checksum in Abbildung 4.8). Falls eine oder mehrere Nachrichten unter der Verwendung von Bestätigungen versendet werden sollen, arbeitet der Sender nach dem ARQ-Prinzip (Automatic Repeat Request [115]), das auch als Send-And-Wait bezeichnet wird. Dies bedeutet, dass der Sender jeweils nur ein Fragment einer Nachricht versendet und mit dem Senden des nächsten Fragments bzw. der nächsten Nachricht wartet, bis die zugehörige Bestätigung empfangen wurde. Falls keine Bestätigung als Antwort auf die Nachricht innerhalb eines Timeouts von 30 ms empfangen wird, schickt der Sender die Nachricht (bzw. das Fragment der Nachricht) erneut. Dieser Vorgang wiederholt sich insgesamt maximal dreimal oder bis eine Bestätigung empfangen wurde. Diese Parameter wurden analog zu den in Abschnitt 5.1.1 vorgestellten Parametern K_D und T_D gewählt. Wird auch beim dritten Mal keine Bestätigung empfangen, wird

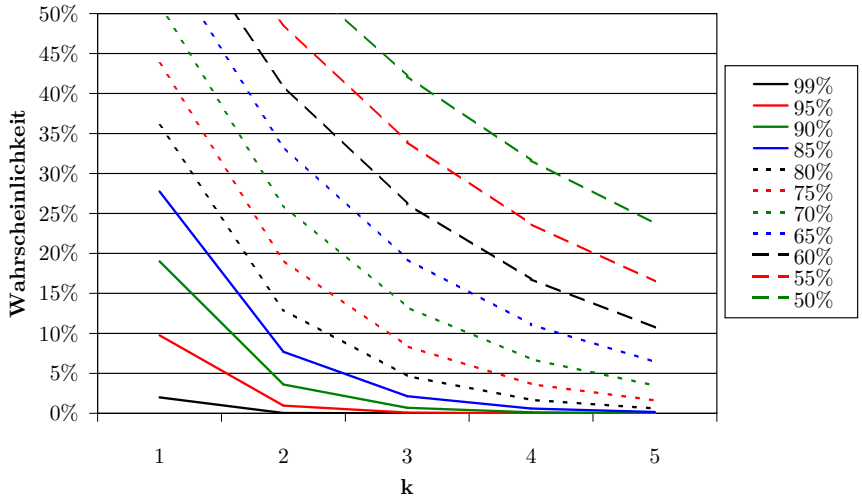


Abbildung 5.3.: Wahrscheinlichkeit, dass eine Nachricht und deren Bestätigung bei k-maligem Senden nicht ankommt (in Abhängigkeit von der Paketankunftsrate)

der Versand der Nachricht gestoppt, auch wenn noch weitere Fragmente der Nachricht nicht gesendet wurden. Anschließend wird der RPC_Message_Handler über den Misserfolg benachrichtigt.

Beim Empfang einer Nachricht wird zunächst der darin enthaltene CCM-MAC auf seine Gültigkeit untersucht. Hierfür berechnet der Empfänger ebenfalls den CCM-MAC der Nachricht unter Verwendung des gemeinsamen Schlüssels K_{CONN} (siehe Abschnitt 5.2). Stimmt der vom Empfänger berechnete CCM-MAC nicht mit dem in der Nachricht beinhalteten CCM-MAC überein, so wird die Nachricht verworfen. In diesem Fall wird keine Bestätigung versendet.

Der in der Nachricht enthaltene Zähler entspricht dem Nachrichtenzähler des Absenders der Nachricht, d.h. es handelt sich hierbei um CNT_{Server} im Falle des Servers und CNT_{Client} im Falle des Clients. Der Nachrichtenzähler wird beim Versenden jeder Nachricht um eins erhöht und hat drei Funktionen

1. Filterung von Duplikaten beim Empfänger der Nachricht
2. Zuordnung der einzelnen Fragmente zu einer gemeinsamen Nachricht
3. Zuordnung einer Antwort/Bestätigung zu einer Nachricht

Beim Empfang einer Nachricht vergleicht der Empfänger den in der Nachricht enthaltenen Zähler (C_{CURR}) mit dem zuletzt empfangenen Zähler (C_{LAST}). Falls $C_{CURR} < C_{LAST}$ gilt, so handelt es sich um eine Nachricht, die außerhalb der Reihenfolge empfangen wurde. Dies kann z.B. auf einen Replay-Angriff durch einen Angreifer zurückzuführen sein. Aufgrund des Send-And-Wait-Prinzips kann dies bei der Verwendung von Bestätigungen nicht auftreten, weshalb in diesem Fall keine Bestätigung versendet wird. Falls $C_{CURR} == C_{LAST}$ gilt, so handelt es sich um ein Duplikat der vorher empfangenen Nachricht und eine Bestätigung wird versendet, falls dies vom Sender angefordert wurde. Dies ist wichtig, da das Duplikat als Reaktion auf eine verlorene Bestätigung in Kombination mit dem Send-and-Wait Prinzip entstanden sein kann: Der Sender sendet eine Nachricht mit Zähler i, der Empfänger empfängt diese Nachricht und sendet eine Bestätigung mit demselben Zähler i zurück. Diese Bestätigung geht verloren. Daraufhin läuft ein Timeout beim Sender ab und dieser Sendet die Nachricht mit Zähler i erneut. Der Empfänger empfängt die Nachricht erneut, sendet eine Bestätigung zurück und verwirft die Nachricht.

Falls $C_{CURR} > C_{LAST}$ gilt, so speichert der Empfänger den in der Nachricht beinhalteten Zähler als C_{LAST} und sendet eine Bestätigung, falls dieses angefordert wurde. Die Nachricht bzw. das Fragment der Nachricht wird dann in der eingehenden Nachrichtenqueue gespeichert. Details zu dieser Queue finden sich in Abschnitt 5.1.4.

5.1.3. RPC-Connection- und Message-Handler

Der Aufbau einer DPS-Verbindung wird vom zugehörigen Client-Stub ausgelöst, der die Verbindung zum Austausch von Nachrichten mit dem Server-Skeleton benötigt. Da der Client-Stub für seine ordnungsgemäße Funktion auf die DPS-Verbindung angewiesen ist, muss er über den aktuellen Zustand der Verbindung informiert werden, um z.B. bei einem Verbindungsabbruch den Aufbau einer neuen Verbindung durch das DPS-Protokoll auszulösen. Dies kann prinzipiell auf zwei Arten geschehen: Reaktiv und proaktiv. Bei einem reaktiven Vorgehen prüft der Client-Stub vor dem Versenden einer RPC-Nachricht an den Server-Skeleton, ob die zugehörige DPS-Verbindung noch aktiv ist. Falls die Verbindung nicht mehr aktiv ist, wird zu diesem Zeitpunkt zunächst eine neue DPS-Verbindung aufgebaut und die RPC-Nachricht in eine Warteschlange eingereiht. Sobald die Verbindung aufgebaut wurde, kann die Nachricht aus der Warteschlange versendet werden. Wird hingegen ein proaktives Vorgehen gewählt, so überwacht der Client-Stub die DPS-Verbindung kontinuierlich und löst einen neuen Verbindungsaufbau aus, sobald die aktuelle Verbindung nicht mehr aktiv ist, unabhängig davon, ob in diesem Moment eine RPC-Nachricht versendet werden soll.

Beide Vorgehensweisen haben Vor- und Nachteile: Das reaktive Vorgehen baut nur dann eine neue DPS-Verbindung auf, wenn diese benötigt wird. Andererseits wird hierdurch allerdings die Latenz beim Versenden der Nachricht erhöht: Vor jedem Sendevorgang muss die Verbindung überprüft werden. Dies kann z.B. durch die Verwendung von Bestätigungen erreicht werden. Falls die zugehörige Verbindung nicht aktiv ist, muss zu diesem Zeitpunkt zunächst eine neue Verbindung aufgebaut werden.

Das proaktive Vorgehen hingegen baut auch dann eine DPS-Verbindung auf, wenn diese nicht benötigt wird. In diesem Fall werden Heartbeat-Nachrichten zur Erkennung eines Verbindungsabbruchs ausgetauscht, obwohl die DPS-Verbindung von der Anwendung nicht zum Versenden von Nachrichten benötigt wird. Dies sorgt folglich für eine nicht notwendige Erhöhung des Nachrichtenaufkommens. Dafür entsteht keine Verzögerung zum Sendezeitpunkt, da die zugehörige Verbindung bereits proaktiv aufgebaut wurde. Da über die DPS-Verbindung nicht nur Nachrichten gesendet werden, sondern auch empfangen werden, kann ein Client nicht entscheiden, ob eine DPS-Verbindung zu diesem Zeitpunkt benötigt wird - stattdessen muss immer eine aktive DPS-Verbindung zur Verfügung stehen.

Um den Zustand einer Verbindung zu beobachten sind ebenfalls zwei Vorgehensweisen denkbar: Polling oder die Verwendung von Events. Beim Polling wird der Status der Verbindung in regelmäßigen Abständen abgefragt, um auf eine Veränderung des Zustands zu reagieren. Bei der Verwendung von Events hingegen registriert sich der Client-Stub als Event-Handler beim DPS-Protokoll, um über alle Veränderungen der Verbindung informiert zu werden. Während das Polling eine Verzögerung in Abhängigkeit von der Periodenlänge besitzt, können Events sofort nach dem Verändern des Status einer Verbindung behandelt wer-

```
1  class DpsEventHandler : public iSenseObject {
2   const uint8 DPS_EVENT_CONNECTION_ESTABLISHED = 60;
3   const uint8 DPS_EVENT_SEND_FAILED = 70;
4   const uint8 DPS_EVENT_SEND_SUCCESS = 71;
5   const uint8 DPS_EVENT_REQUEST = 72;
6   const uint8 DPS_EVENT_RESPONSE = 73;
7   const uint8 DPS_EVENT_CONNECTION_RESET = 74;
8
9   virtual void handleDpsConnectionEvent(uint8 event_type,
10                            DpsConnection* connection) = 0;
11
12  virtual void handleDpsRpcCallEvent(uint8 event_type,
13                            uint32 event_id,
14                            DpsRpcMessage* message) = 0;
15 };
```

Abbildung 5.4.: Schnittstellen des DPS-Event-Handlers

den. Außerdem wird hierbei eine standardisierte Schnittstelle geschaffen, die von allen Client-Stubs und Server-Skeletons verwendet werden kann, während das Polling getrennt implementiert werden muss. Auch beim Versenden von RPC-Nachrichten kann eine event-basierte Vorgehensweise verwendet werden. So kann z.B. der Client-Stub bei der Verwendung von Bestätigungen (siehe Abschnitt 4.1.4) über den erfolgreichen Versand einer Nachricht informiert werden oder im Falle eines nicht erfolgreichen Versands in Abhängigkeit des verwendeten Protokolls reagieren.

Aus den oben genannten Gründen hat sich der Autor bei der Implementierung des DPS-Protokolls für eine proaktive, event-basierte Vorgehensweise entschieden. Hierfür wurde die Klasse `DpsEventHandler` entwickelt, deren Schnittstellen in Abbildung 5.4 dargestellt sind, die von einem Client-Stub oder Server-Skeleton implementiert werden müssen. Die Methode `handleDpsConnectionEvent()` erhält als Parameter den Typ des Ereignisses sowie die zugehörige Verbindung und wird vom DPS-Protokoll aufgerufen, sobald eine Verbindung aufgebaut wurde (`DPS_EVENT_CONNECTION_ESTABLISHED`) oder ein Verbindungsabbruch erkannt wurde (`DPS_EVENT_CONNECTION_RESET`).

Die Methode `handleDpsRpcCallEvent()` hingegen wird beim Senden und Empfangen von RPC-Nachricht verwendet. Die Methode erhält als Parameter den Typ des Ereignisses, die Event-ID sowie die betroffene RPC-Nachricht. Beim Senden einer RPC-Nachricht erhält der Sender eine Event-ID zurück, die dem lokalen Nachrichtenzähler entspricht, der in der Nachricht enthalten ist (Counter in Abbildung 4.12). Falls diese Nachricht ohne Bestätigungen versendet wird, erhält der Sender ein Callback vom Typ `DPS_EVENT_SEND_SUCCESS`, sobald die RPC-Nachricht die Nachrichtenqueue (siehe Seite 77) verlassen hat und gesendet wurde. Wird die RPC-Nachricht hingegen unter Verwendung von Bestätigungen versendet, erhält der Sender das Callback erst nach dem erfolgreichen Empfang der zugehörigen Bestätigung vom Empfänger. Wird hingegen auch nach dem letzten Sendeversuch keine Bestätigung empfangen, erhält der Sender ein Ereignis vom Typ `DPS_EVENT_SEND_FAILED` und muss dieses behandeln.

Beim Empfang einer RPC-Nachricht erhält der Empfänger ein Ereignis vom Typ `DPS_EVENT_REQUEST` und kann anschließend die Nutzlast der empfangenen RPC-Nachricht deserialisieren. Falls es sich bei der empfangenen RPC-Nachricht um eine Antwort auf einen von diesem Knoten gesendeten RPC-Request handelt, hat das Ereignis den Typ `DPS_EVENT_RESPONSE` und kann mittels der übergebenen Event ID dem entsprechenden Anfrage zugeordnet werden, da die Event ID identisch ist.

5.1.4. Nachrichtenqueue

Beim Senden von Nachrichten kann es passieren, dass die Ausgabegeschwindigkeit des Protokolls, das Nachrichten über das DPS-Protokoll versenden möchte, die Sendegeschwindigkeit der Funkschnittstelle überschreitet. Auf diese Situation kann potentiell auf mehrere Arten reagiert werden: Falls das DPS-Protokoll bereits mit dem Senden einer Nachricht beschäftigt ist, wird entweder die neu eintreffende Nachricht verworfen oder aber die vorherige Nachricht ersetzt. In beiden Fällen kommt es zu einem Nachrichtenverlust, da eine der beiden Nachrichten nicht vollständig gesendet werden kann. Eine weitere Alternative besteht darin, eine Fehlermeldung an das sendende Protokoll zu melden, wodurch dieses den Fehler behandeln muss. Dies kann z.B. durch das erneute Senden der Nachricht zu einem späteren Zeitpunkt geschehen, was in der Form einer Warteschlange implementiert werden kann. Da das DPS-Protokoll von mehreren Protokollen verwendet werden kann, sollte diese Warteschlange in das DPS-Protokoll integriert werden, um eine mehrfache und eventuell fehlerhafte Implementierung durch die aufrufenden Protokolle zu vermeiden.

Aus diesem Grund hat sich der Autor dazu entschlossen, eine Warteschlange für den Nachrichtenversand zu verwenden, die vom DPS-Protokoll verwaltet wird. Diese Nachrichtenqueue kann mehrere RPC-Nachrichten speichern, wobei die maximale Anzahl vom verfügbaren Speicher auf dem Sensorknoten abhängt. Die Nachrichtenqueue speichert Pointer auf Objekte vom Typ RpcMessage sowie die aktuelle Lese- und Schreibposition. Abbildung 5.5 zeigt als Beispiel eine Nachrichtenqueue mit insgesamt sechs RPC-Nachrichten, die in der Queue gespeichert sind. Die Nachricht wird zunächst in der Queue gespeichert, die Schreibposition wird um eins erhöht und das DPS-Protokoll wird benachrichtigt. Falls das DPS-Protokoll zu diesem Zeitpunkt bereits mit dem Senden einer anderen Nachricht beschäftigt ist, passiert nichts. Ist die restliche Queue leer, so beginnt das DPS-Protokoll mit dem Versenden der neuen Nachricht. Hierfür wird der im nächsten Unterabschnitt beschriebene Fragmentierungsmechanismus verwendet. Nachdem eine RPC-Nachricht verschickt wurde, erhöht das DPS-Protokoll den Lesekopf um eins und prüft, ob noch weitere Nachrichten in der Queue vorhanden sind. Falls dem so ist, fährt das DPS-Protokoll solange mit dem Senden fort, bis die Queue leer ist. Die Elemente der Queue werden nach dem FIFO-Prinzip (First-In, First-Out) gesendet, d.h. sie werden in der Reihenfolge gesendet, in der Sie eingefügt worden sind.

In Abhängigkeit von der Größe der Nachrichtenqueue und der Sendegeschwindig-

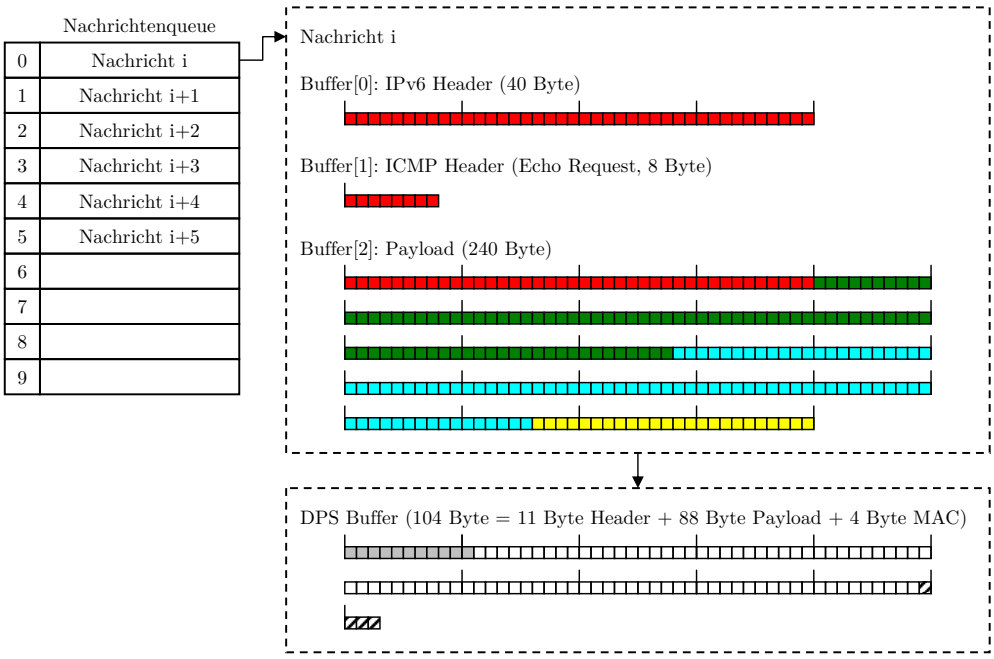


Abbildung 5.5.: Nachrichtenqueue mit vier darin gespeicherten Nachrichten und Darstellung des Fragmentierungsmechanismus am Beispiel einer Nachricht aus vier Fragmenten (Farbliche Markierung)

keit kann es auch in diesem Fall dazu kommen, dass weitere RPC-Nachrichten in eine volle Queue eingefügt werden sollen. In diesem Fall werden die vorhandenen Nachrichten durch neu eintreffende Nachrichten überschrieben.

5.1.5. Fragmentierungsmechanismus

Beim Versenden von Nachrichten übernimmt das DPS-Protokoll die Fragmentierung der Nachricht, die den in Abschnitt 4.1.3 beschriebenen Mechanismus verwendet. Hierbei wird die Nachricht in Fragmente unterteilt, die dann mittels mehrerer RPC-Nachrichten vom Sender zum Empfänger verschickt und dort wieder zusammengesetzt werden. Ein Beispiel für eine solche Nachricht findet sich in Abbildung 5.5: In diesem Beispiel soll eine ICMP-Echo-Request-Nachricht mit 240 Byte Payload verschickt werden. Hieraus ergibt sich eine Gesamtgröße von $n = 288$ Byte für die zu versendende IPv6-Nachricht (40 Byte IPv6-Header, 8 Byte ICMP-Header, 240 Byte ICMP-Payload), die vom DPS-Protokoll in insgesamt vier RPC-Nachrichten verschickt wird.

Der IP-Stack des iSense-Betriebssystems verwaltet IPv6-Nachrichten als Buffer-Array, das im Beispiel in Abbildung 5.5 eine Länge von drei aufweist. Jeder Eintrag dieser Arrays beinhaltet hierbei den Header und/oder Payload einer anderen Schicht des Protokollstapels, wie in Abbildung 5.5 zu sehen ist: Element 0 des Arrays beinhaltet den IPv6-Header, Element 1 den ICMP-Header und Element 2 den ICMP-Payload der Nachricht. Bei der Serialisierung der IPv6-Nachricht durch das DPS-Protokoll sind mehrere Vorgehensweisen möglich, die

im Folgenden beschrieben und hinsichtlich ihres Speicherverbrauchs verglichen werden. Der Speicherverbrauch des ursprünglichen Buffer-Array wird hierbei mit n bezeichnet.

Eine Möglichkeit besteht darin, das Buffer-Array in ein zusammenhängendes Array A zu linearisieren, das somit die Konkatenation der einzelnen Elemente des Buffer-Arrays beinhaltet. Aus diesem Array können dann die einzelnen RPC-Fragmente erstellt werden. Im schlechtesten Fall kann aus dieser Vorgehensweise eine Verdreifachung des Speicherbedarfs auf $(3 * n) + c$ resultieren, da der Speicherinhalt des Buffer-Arrays zunächst in das Array A und anschließend in die einzelnen RPC-Fragmente kopiert wird. Hierbei entspricht c der Größe des DPS-Buffers (104 Byte) plus der Größe der restlichen verwendeten Datenstrukturen. Als ein naheliegender Optimierungsschritt kann das Buffer-Array nach der Linearisierung gelöscht werden. Somit wird die Fragmentierung der einzelnen Elemente nicht in einem Schritt durchgeführt, sondern erst zum Zeitpunkt des Sendevorgangs des entsprechenden Fragments. Auf diese Weise kann der Speicherverbrauch gesenkt werden, er beträgt jedoch durch die Linearisierung des Buffer-Arrays zumindest für einen kurzen Zeitraum $(2 * n) + c$.

Um den Speicherverbrauch weiter zu reduzieren hat sich der Autor für einen weiteren Optimierungsschritt entschieden, der auf die Linearisierung des Buffer-Arrays verzichtet. Hierfür arbeitet auch das DPS-Protokoll intern direkt mit den Buffer-Arrays des iSense-IPv6-Stack, deren Inhalt erst beim Sendevorgang eines Fragments in den DPS-Buffer kopiert wird. Wie in Abbildung 5.5 zu sehen ist, beinhaltet jede RPC-Nachricht in der Sending-Queue einen Pointer auf das zu sendende Buffer-Array. Die im Beispiel angegebene RPC-Nachricht besitzt eine Länge von $n = 288$ Byte und muss somit auf insgesamt vier RPC-Fragmente aufgeteilt werden[1]. Die einzelnen Fragmente sind in Abbildung 5.5 farblich markiert. Beim ersten Sendevorgang kopiert das DPS-Protokoll die rot markierten Byte aus dem IPv6-Header, dem ICMP-Header und dem Payload in den Buffer des DPS-Fragments. Zusammen mit dem DPS-Header und der Prüfsumme ergibt sich somit eine Fragmentlänge von 104 Byte (siehe Abschnitt 4.2). Nachdem dieses Fragment über die Funkschnittstelle gesendet wurde und der entsprechende Speicher wieder freigegeben wurde, beginnt das DPS-Protokoll mit dem Senden des nächsten Fragments. Hierfür werden die grün markierten Bytes in den DPS-Buffer kopiert. Dieser Vorgang wiederholt sich für die blau und gelb markieren Bytes. Anschließend wird auch der Speicher des Buffer-Arrays und der RPC-Nachricht freigegeben, die Queue an der Leseposition mit NULL überschrieben und die Leseposition um eins erhöht. Durch die in diesem Absatz beschriebene Vorgehensweise kann der Speicherverbrauch auf $n + c$ reduziert werden. Ein Nachteil dieser Vorgehensweise ist, dass der Speicher der bereits gesendeten Fragmente erst wieder freigegeben wird, sobald das letzte Fragment gesendet worden ist.

[1] Es können in diesem Beispiel maximal 88 Byte pro Fragment versendet werden, da keine Header-Kompression verwendet wird und die Länge der Prüfsumme 4 Byte beträgt.

5.1.6. Erkennung von Verbindungsabbrüchen

Die Erkennung von Verbindungsabbrüchen erfolgt mittels des in Abschnitt 4.1.5 vorgestellten Mechanismus, der den periodischen Austausch von Heartbeat-Nachrichten zwischen den beiden Kommunikationspartnern der DPS-Verbindung vorsieht. Jeder Kommunikationspartner führt hierbei in der Datenstruktur, die für die Speicherung der Verbindungsdaten zuständig ist, zwei Zeitstempel mit: T_{send} gibt den Zeitpunkt der letzten versendeten Nachricht und T_{rec} den Zeitpunkt der letzten empfangenen Nachricht für diese Verbindung an. Diese Zeitstempel werden bei jedem erfolgreichen Empfangsereignis und bei jedem Sendeereignis mit dem aktuellen lokalen Zeitstempel des Sensorknotens aktualisiert. In diesem Abschnitt werden zunächst die Funktionsweise und das Zusammenspiel dieser Parameter beschrieben und abschließend auf die Wahl der Werte für diese Parameter eingegangen.

Um zu erkennen, wann ein Verbindungsabbruch vorliegt oder wann eine Heartbeat-Nachricht gesendet werden muss, werden die beiden Zeitstempel T_{send} und T_{rec} für jede aktive DPS-Verbindung regelmäßig überprüft. Falls T_{send} älter als I_{send} ist, d.h. wenn dieser Knoten keine Nachricht an seinen Kommunikationspartner innerhalb der letzten I_{send} ms geschickt hat, sendet dieser Knoten eine Heartbeat-Nachricht und aktualisiert T_{send} mit dem aktuellen lokalen Zeitstempel. Falls hingegen T_{rec} älter als I_{rec} ist, d.h. wenn dieser Knoten keine Nachricht von seinem Kommunikationspartner innerhalb der letzten I_{rec} ms erhalten hat, wird die zugehörige DPS-Verbindung als inaktiv betrachtet und gelöscht. Falls es sich bei diesem Knoten um einen Client handelt, beginnt der Discovery-Prozess erneut, damit der Client nach einem neuen Server für das entsprechende Protokoll suchen kann.

Die Parameter I_{send}, der die Häufigkeit des Sendens einer Heartbeat-Nachricht steuert, und I_{rec}, der die Zeit steuert, nach der eine Verbindung ohne Empfang einer Nachricht getrennt wird, können in Abhängigkeit vom Einsatzszenario und den Anforderungen der Anwendung angepasst werden. Hierbei ist prinzipiell ein Kompromiss zwischen Erkennungsdauer und Falsch-Positiv-Rate notwendig. Soll ein Verbindungsabbruch möglichst schnell erkannt werden, so sollte I_{rec} möglichst gering gewählt werden. Gleichzeitig muss immer die Bedingung $k * I_{send} < I_{rec}$ (mit $k > 1{,}0$) gelten, da sonst das Sendeintervall der Heartbeat-Nachrichten größer als das Erkennungsintervall ist. Der Parameter k sollte in diesem Fall nicht zu klein gewählt werden bzw. I_{send} sollte nicht zu groß gewählt werden, um die Falsch-Positiv-Rate zu reduzieren. Im Betrieb des Sensornetzes gehen Heartbeat-Nachrichten auch durch Nachrichtenverlust verloren, ohne dass ein Verbindungsabbruch vorliegen muss. Dies kann z.B. durch Kollisionen oder Übertragungsfehler verursacht werden.

Um einen Anhaltspunkt für die Wahl des richtigen Verhältnisses zwischen den beiden Parameter I_{send} und I_{rec} zu erhalten, wird im Folgenden eine theoretische Analyse der Falsch-Positiv-Rate angestellt, welche die bereits in Abschnitt 5.1.1 vorgestellten Untersuchungen zum Nachrichtenverlust verwendet. Die Falsch-Positiv-Rate hängt von zwei Faktoren ab: Die durchschnittliche Paketankunfts-

rate des Funkmediums sowie das Verhältnis zwischen I_{send} und I_{rec}, das mit dem Parameter k gesteuert wird.

Bei einer durchschnittlichen Paketankunftsrate P_{rec} gehen Nachrichten mit einer Wahrscheinlichkeit von $P_{loss} = 1 - P_{rec}$ auf dem Funkmedium verloren. Ein Verbindungsabbruch wird erkannt, wenn mindestens k direkt aufeinander folgende Heartbeat-Nachrichten verloren gehen. Die Wahrscheinlichkeit $P_{false\_positive}(P_{rec}, k)$ kann hierfür nach der folgenden Formel berechnet werden:

$$P_{false\_positive}(P_{rec}, k) = (1 - P_{rec})^k \tag{5.3}$$

Dies entspricht der in Abschnitt 5.1.1 vorgestellten Formel für die kumulative Nachrichtenverlustwahrscheinlichkeit. Die Ergebnisse dieser Formel für unterschiedliche Nachrichtenverlustwahrscheinlichkeiten ist in Abbildung 5.2 auf Seite 72 dargestellt. Die kumulative Wahrscheinlichkeit entspricht hierbei der in diesem Abschnitt gesuchten Falsch-Positiv-Rate $P_{false\_positive}(P_{rec}, k)$. Soll z.B. bei einer durchschnittlichen Paketankunftsrate von 70 % eine Falsch-Positiv-Rate von $P_{false\_positive} < 1\,\%$ erzielt werden, so muss $k \geqslant 4$ gewählt werden. Dies ist z.B. mit den Werten I_{send}=1000 ms sowie I_{rec}=4000 ms möglich.

5.2. Berechnung des MAC

Die Berechnung des MAC einer Nachricht erfolgt mittels des AES-CCM-MAC-Algorithmus, also einer Kombination aus dem AES-Counter-Modus und einem CBC-MAC (siehe RFC 3610 [116]). Unter einem CBC-MAC (cipher block chaining - message authentication code) versteht man einen MAC, der durch die Verkettung einer Blockchiffre entsteht. AES-CCM-MAC wurde zur Berechnung des MAC gewählt, da die verwendeten Sensorknoten vom Typ JN5139 und JN5148 die hardwarebeschleunigte Berechnung dieses Algorithmus unterstützen. Dies trifft auf alle Sensorknoten zu, die ein 802.15.4-Funkmodul aufweisen, da der zugehörige Standard die Unterstützung dieses Verschlüsselungsverfahrens vorschreibt.

Für die Berechnung des CBC-MAC wird die Nachricht M zunächst in n Blöcke der Größe 16 Byte (128-Bit) aufgeteilt (B_1 bis B_n), wie in Abbildung 5.6 dargestellt. Der Block B_0 beinhaltet die Nonce, die Länge des Klartextes und einige Flags. Falls die Länge der Nachricht M kein Vielfaches der Blockgröße ist, wird der letzte Block mit Nullen aufgefüllt, bis dieser die notwendige Länge aufweist (Zero-Padding). Jeder dieser Blöcke wird mittels der Blockchiffre AES und einem geheimen Schlüssel K verschlüsselt. Das Ergebnis der Verschlüsselung eines Blocks fließt in die Verschlüsselung des nächsten Blocks ein, indem das Ergebnis mit dem nächsten Block der Nachricht XOR verknüpft wird, bevor dieser verschlüsselt wird. Auf diese Weise entsteht ein 16 Byte langer MAC, der durch das Weglassen der niedrigstwertigen Bits in einen MAC der Länge 2, 4 oder 8 Byte transformiert werden kann.

Dies wird in [10] (Annex B: CCM* mode of operation) genauer beschrieben. Die Berechnung des CCM-MAC beinhaltet hierbei noch weitere Schritte, die zur Transformation der Eingabedaten vor der eigentlichen Berechnung des MAC durchgeführt werden. Diese sind in [10] (B.4.1.1/B.4.1.2) genauer beschrieben und beinhalten unter anderem die Aufnahme der Nachrichtenlänge sowie der Reserved- und AData-Felder in die Berechnung des MAC durch die Verkettung dieser Werte mit dem Klartext der Nachricht vor der Blockbildung. Die Nachrichtenlänge wird in die Berechnung aufgenommen, um Nachrichten mit beliebiger Länge zu schützen, da die Sicherheit von CCM-MACs nur für Nachrichten mit fester Länge gilt [117].

Der verwendete CBC-MAC besitzt eine Länge von 4 Byte, was laut [113] und [118] ausreichend für ein drahtloses Sensornetz ist: Der Angreifer kann das zugrundeliegende Verschlüsselungsverfahren nicht brechen und besitzt keinen Zugriff auf das Schlüsselmaterial (siehe Abschnitt 4.3.1). Um also einen gültigen MAC für eine von ihm erzeugte Nachricht zu erzeugen, muss er alle 2^{32} möglichen Kombinationen ausprobieren. Da er keine Möglichkeit besitzt, die Gültigkeit des MAC selbst zu überprüfen, muss er die erzeugten Nachrichten an einen Sensorknoten im Netzwerk schicken, um die Gültigkeit des MAC festzustellen. Laut der Autoren aus [113] kann ein Angreifer in einem Sensornetz mit einer verfügbaren Bandbreite von 19,2kB/s maximal 40 Nachrichten pro Sekunde verschicken, so dass das Erraten eines gültigen MACs der Länge 4 Byte etwa 20 Monate dauert. Dies übersteigt die gewöhnliche Lebensdauer eines batteriebetriebenen Sensornetzes und erlaubt es dem Angreifer lediglich, eine einzige gefälschte Nachricht einzuschleusen.

Falls eine Verwendung des AES-CCM-MAC-Algorithmus auf einer Sensornetzplattform oder für ein gegebenes Einsatzgebiet nicht möglich oder gewünscht ist, kann die MAC auch mit anderen Algorithmen berechnet werden oder es kann auf den Einsatz einer Prüfsumme verzichtet werden, falls dies vom Sensornetzbetreiber gewünscht ist und die Anwendungsanforderungen dies zulassen. Falls die Berechnung einer Prüfsumme gewünscht ist, aber keine Authentifizierung benötigt wird, kann statt AES-CCM-MAC auch MD-5, SHA-1 oder HMAC verwendet werden. Für Testzwecke wurde deshalb zusätzlich zu der AES-CBC-

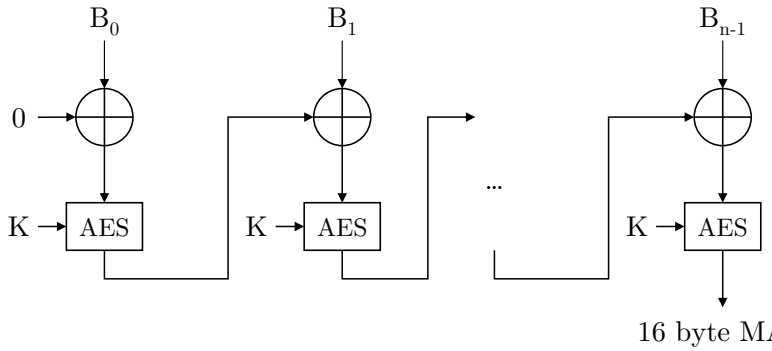


Abbildung 5.6.: Anwendung des CBC-MAC-Algorithmus auf die Blöcke B_0 bis B_{n-1} einer Nachricht M

MAC basierenden Prüfsumme eine XOR-basierende Prüfsumme implementiert, die durch die XOR-Verknüpfung des Nachrichteninhalts mit dem Verbindungsschlüsseln in Blöcken der Länge der gewünschten Prüfsummenlänge berechnet wird.

5.2.1. Verwendete Schlüssel

Das DPS-Protokoll verwendet zur Berechnung des MAC zwei verschiedene Schlüssel: K_{DISC} und K_{CONN}. Für die Berechnung des MAC während des Discovery-Prozesses wird der globale Schlüssel K_{DISC} verwendet, da in dieser Protokollphase Broadcast-Kommunikation zum Einsatz kommt und somit kein paarweiser Schlüssel zwischen dem Sender und Empfänger verwendet werden kann. K_{DISC} muss allen Sensorknoten des Netzwerks bekannt sein und kann entweder innerhalb des Programmcodes festgelegt sein oder durch ein Schlüsselverteilungsprotokoll verteilt werden.

Während des Verbindungsaufbaus und des anschließenden Nachrichtenaustausches verwenden der Server und der Client den Verbindungsschlüssel K_{CONN}, der aus dem paarweisen Schlüssel zwischen Server und Client K_{PAIR} und der Verbindungs-Nonce N_{CONN} gebildet wird. Zur Berechnung von K_{CONN} wird dieser zunächst in vier Teile mit jeweils 32-Bit aufgeteilt, die jeweils mit der 32-Bit Verbindungs-Nonce initialisiert werden. Anschließend wird K_{CONN} mittels K_{PAIR} verschlüsselt, wodurch sich der einmalige Verbindungsschlüssel ergibt.

N_{CONN} wird vom Client zufällig erzeugt und anschließend mit der ersten Nachricht des Verbindungsaufbaus an den Server geschickt. Der paarweise Schlüssel K_{PAIR} hingegen darf ausschließlich den beiden Kommunikationspartnern bekannt sein und muss daher mittels eines Schlüsselverteilungsprotokolls verteilt werden (z.B. HARPS [112]). Da für jede neue Verbindung zwischen Server und Client eine neue Verbindungs-Nonce vom Client erstellt wird, muss K_{PAIR} nicht für jede neue Verbindung aktualisiert werden, sondern kann für bis zu 2^{32} Verbindungen verwendet werden. Hierbei ist jedoch sicherzustellen, dass jede Nonce vom Client nur für genau einmal mit demselben K_{PAIR} für eine Verbindung genutzt wird.

Zusätzlich wird noch ein privater Schlüssel für den sicheren Pseudo-Zufallszahlengenerator verwendet, der ausschließlich dem jeweiligen Sensorknoten bekannt sein darf. Der Verwendungszweck dieses Schlüssels wird im nachfolgenden Abschnitt genauer beschrieben.

5.3. Pseudo-Zufallszahlengenerator

Während des Protokollablaufs müssen mehrere Zufallszahlen sowohl vom Client als auch vom Server erzeugt werden (N_{CONN}, CNT_{Server} und CNT_{Client}). Hierfür können vom DPS-Protokoll zwei unterschiedliche Zufallszahlengeneratoren verwendet werden, die ausgehend von einem Startwert (Seed) eine Reihe von

Zufallszahlen erzeugen. Bei Zufallszahlengeneratoren werden generell zwei Kategorien unterschieden: Nicht-deterministische Zufallszahlengeneratoren (RNG) und deterministische Pseudo-Zufallszahlengeneratoren (PRNG). Während ein RNG immer eine andere Folge von Zufallszahlen erzeugt, erzeugt ein PRNG ausgehend vom gleichen Seed immer dieselbe Folge von Zufallszahlen. Da Software immer deterministisch arbeitet, ist für die Implementierung eines RNG die Verwendung einer zusätzlichen Hardwarekomponente notwendig. Im Rahmen des DPS-Protokolls wird zur Erzeugung von (unsicheren) Zufallszahlen ein PRNG auf Basis eines linearen Kongruenzgenerators verwendet (LCG [119]). Falls das DPS-Protokoll sichere Zufallszahlen erzeugen muss, z.B. in Gegenwart eines potentiellen Angreifers, wird der sichere PRNG verwendet, der im nachfolgenden Unterabschnitt beschrieben wird. In diesem Abschnitt erfolgt zunächst eine Beschreibung des LCG. Ein LCG berechnet ausgehend von seinem Seed (y_0) eine Folge von Zufallszahlen mittels der folgenden Formel:

$$y_{i+1} = (a * y_i + b) \ mod \ m \tag{5.4}$$

Der LCG erzeugt eine Zufallszahl auf Basis seines internen Zustands y_i. Hierfür wird der interne Zustand des Generators mit a multipliziert und anschließend b hinzuaddiert. Anschließend wird die Modulo-Operation auf diese Zahl angewendet, wodurch sich der neue Zustand y_{i+1} ergibt. Die resultierende Zufallszahl entspricht dem internen Zustand modulo dem gewünschten Zahlenbereich.

Die Parameter des LCG werden für das DPS-Protokoll hierbei mit $a = 1664525$, $b = 1013904223$ und $m = 2^{32}$ initialisiert, die von Numerical Recipes [120] vorgeschlagen werden. Durch die Verwendung von $m = 2^{32}$ kann die Modulo-Operation eingespart werden, wenn für y_i eine 32-Bit Variable verwendet wird. Dies ist keine Einschränkung für das DPS-Protokoll, da die erzeugten Zufallszahlen ohnehin als 32-Bit Werte verwendet werden.

5.3.1. Sicherer Pseudo-Zufallszahlengenerator

Der LCG ist sehr einfach zu implementieren, schnell und benötigt wenig Speicherplatz. Er erzeugt jedoch keine sichere Folge von Zufallszahlen, da der Dolev-Yao-Angreifer durch die Kenntnis der Parameter a, b und m aus jeder Zufallszahl alle folgenden Zufallszahlen berechnen kann. Aus diesem Grund sollte ein LCG nur dann eingesetzt werden, wenn das Anwendungsszenario dies zulässt. In diesem Abschnitt werden deshalb die Mechanismen beschrieben, mit denen ein sicherer Zufallszahlengenerator für das DPS-Protokoll implementiert wurde.

Da die erzeugte Folge von Zufallszahlen bei der Verwendung desselben Seeds immer identisch ist und das Setzen des Seeds des PRNG in der Standardkonfiguration nur beim Boot des Sensorknoten erfolgt, ergeben sich mehrere Probleme für den Betrieb des DPS-Protokolls: Wird der PRNG beim Boot des Sensorknoten immer mit demselben Seed initialisiert (z.B. ein fester Wert oder die MAC-Adresse des Sensorknoten), so wird nach jedem Bootvorgang dieselbe

Folge von Zufallszahlen erzeugt. Da der Client beim Verbindungsaufbau jedoch die Nonce N_{CONN} generieren muss, die nur genau einmal im Zusammenhang mit dem paarweisen Schlüssel zwischen Client und Server verwendet werden darf (siehe Abschnitt 4.3.3), ergibt sich hierdurch eine Sicherheitslücke, die von einem potentiellen Angreifer ausgenutzt werden kann.

Des Weiteren ist es möglich, dass ein Angreifer die Nachrichtenzähler CNT_{Server} und CNT_{Client} im Rahmen eines Replay-Angriffs auf das Protokoll zur Verbindungsüberwachung (siehe Abschnitt 4.1.5) nutzt. Dies ist z.B. möglich, wenn dieselben Zähler in einer vorherigen Verbindung zusammen mit derselben Nonce verwendet wurden. In diesem Fall kann der Angreifer die Heartbeats einer vorherigen Verbindung aufzeichnen und bei einer späteren Verbindung (z.B. nach einem Neustart eines Sensorknoten) verwenden.

Bei der Implementierung des sicheren Pseudo-Zufallszahlengenerator müssen also zwei Kriterien erfüllt sein:

- Zufällige Initialisierung: Es darf für einen Angreifer nicht möglich sein, die erste Zufallszahl z_0 durch Kenntnis des Programmcodes oder Manipulation der Umgebung zu berechnen.
- Nicht Vorhersehbarkeit: Es darf für einen Angreifer nicht möglich sein, auf Basis der Zufallszahl z_i eine der nachfolgenden Zufallszahlen z_j (mit $j > i$) zu berechnen.

Für die Erfüllung des ersten Kriteriums muss sichergestellt werden, dass die Initialisierung des PRNG unter Verwendung zufälliger Quellen geschieht, bevor die Nonce und die beiden Counter berechnet werden. Hierfür können die Metadaten der während des Discovery-Prozesses empfangenen Nachrichten verwendet werden - z.B. LQI, RSSI und Zeitstempel. In der Arbeit von [121] wird zusätzlich noch die Verwendung der Bitfehler empfohlen, da diese im Gegensatz zu den vorher genannten Werten deutlich weniger genau von einem Angreifer gemessen oder vorhergesagt werden können. Eine Verwendung der Bitfehler ist auf der iSense-Plattform nicht möglich, da keine Schnittstelle existiert, um auf die Bitfehler der empfangenen Nachrichten zuzugreifen. Der genaue Ablauf dieses Vorgangs ist in Abbildung 5.7 dargestellt: Die bei jedem empfangenen Paket gesammelten Metadaten (LQI, RSSI und Empfangszeitpunkt) werden in einem Entropie-Akkumulator (EA) gesammelt. Hierbei werden die Metadaten abwechselnd mit den Positionen 0 bis 7 und 8 bis 15 des 16 Byte langen EA XOR-verknüpft. Anschließend wird der EA zur Erhöhung der Entropie zusätzlich mittels des CCM-MAC-Modus des AES-Koprozessors verschlüsselt, der auch für die restlichen Sicherheitsmechanismen des DPS-Protokolls verwendet wird. Als Schlüssel dient ein privater Schlüssel, der nur diesem Sensorknoten bekannt sein darf. Zusätzlich wird ein Zähler (Counter) als Nonce verwendet, der bei jeder Anwendung des CCM-MAC um eins erhöht wird. Auf diese Weise wird das Ergebnis der MAC-Berechnung auch dann unterschiedlich ausfallen, wenn dieselben Eingabedaten verwendet werden (was z.B. von einem Angreifer hervorgerufen werden könnte).

Für die Erfüllung des zweiten Kriteriums muss verhindert werden, dass ein Angreifer durch Kenntnis der Implementierung und einer beliebigen Zufallszahl eine oder mehrere zukünftige Zufallszahlen berechnen kann. Da der Inhalt des EA nur beim Empfang eines weiteren Pakets verändert wird, kann ein Angreifer durch das Blockieren des Nachrichtenempfangs eines Knotens den Inhalt des EA einfrieren. Aus diesem Grund wird vor dem Berechnen einer Zufallszahl der CCM-MAC-Modus des AES-Koprozessors verwendet, um eine neue Zufallszahl zu erzeugen. Die Verwendung eines CCM-MAC zur Realisierung eines sicheren PRNG wurde von Jonsson in [122] vorgeschlagen und bewiesen.

Der resultierende CPRNG (cryptographic pseudo-random number generator) läuft nach demselben Schema ab wie TinyRNG [121], der jedoch statt AES-CCM den Skipjack-Algorithmus verwendet. Dieses Schema wurde von Kelsey et al. unter der Bezeichnung Yarrow-160 [123] ursprünglich für SHA-1 und Blowfish konzipiert. TinyRNG nutzt als Entropie-Quelle die Bitfehler von Nachrichten und verwendet Skipjack im CBC-MAC-Modus zur Entropieerhöhung, während Skipjack im Blockchiffre-Modus mit Counter zur Erzeugung von Zufallszahlen verwendet wird. Wie Jonsson in [122] beschreibt, liegt dies daran, dass Blockchiffren nur Verschlüsselung anbieten, während für die Authentifizierung, die bei der Entropieerhöhung im EA benötigt wird, eine kryptographische Hashfunktion (z.B. SHA-1) oder ein CBC-MAC verwendet wird. Ein Beispiel hierfür ist laut [122] SSL/TLS, das Triple-DES als CBC-Blockchiffre mit HMAC kombiniert. AES-CCM kombiniert diese beiden Verfahren auf sichere Art und Weise miteinander, so dass AES-CCM für beide Schritte verwendet werden kann.

5.4. Nachrichtenkompression

Neben der in Abschnitt 4.2.1 beschriebenen Kompression des Headers besteht die Möglichkeit, den Payload der Nachricht zu komprimieren. Im Gegensatz zur Header-Kompression kann hierbei kein Wissen über den Aufbau des Payloads verwendet werden, so dass ein allgemeingültiges Kompressionsverfahren eingesetzt werden muss.

Die Aufgabe eines Kompressionsverfahrens ist es in diesem Fall, den Payload einer Nachricht in eine günstigere Repräsentation zu überführen, so dass der Anteil an redundanter und irrelevanter Information verringert wird. Anschließend soll der komprimierte Payload wieder zurück in den unkomprimierten Payload transformiert werden können. Hierbei werden zwei Klassen von Kompressionsalgorithmen unterschieden: Die verlustfreie Kompression und die verlustbehaftete Kompression. Bei der verlustfreien Kompression ist kein Unterschied zwischen dem ursprünglichem Payload und dem dekomprimierten Payload feststellbar. Bei der verlustbehafteten Kompression hingegen ist ein Unterschied feststellbar. Da das DPS-Protokoll für beliebige Kommunikationsprotokolle einsetzbar sein soll, deren Nachrichtenformat unbekannt ist, verbietet sich der Einsatz von verlustbehafteter Kompression.

Auf dem Gebiet der verlustfreien Kompression existieren einige wohlbekannte

```
uint32 counter = 0;
uint8 entropy_accumulator[16];
uint8 base = 0;

void receive(uint8 len, const uint8* buf, uint16 lqi,
                              uint16 rssi, Time rx_time) {

 accumulate_entropy(lqi, rssi, rx_time.to_millis());
 // Process packet
 // [...]
}


void accumulate_entropy(uint16 lqi, uint16 rssi, uint32 time) {

 entropy_accumulator[base+0] ^= (lqi >> 8) & 0xFF;
 entropy_accumulator[base+1] ^= (lqi >> 0) & 0xFF;
 entropy_accumulator[base+2] ^= (rssi >> 8) & 0xFF;
 entropy_accumulator[base+3] ^= (rssi >> 0) & 0xFF;
 entropy_accumulator[base+4] ^= (time >> 24) & 0xFF;
 entropy_accumulator[base+5] ^= (time >> 16) & 0xFF;
 entropy_accumulator[base+6] ^= (time >> 8) & 0xFF;
 entropy_accumulator[base+7] ^= (time >> 0) & 0xFF;

 base = (base + 8) % 16;

 os_.aes().ccm_mac(  16, entropy_accumulator,
                     16, entropy_accumulator,
                     NODE_PRIVATE_KEY, counter++);
}


uint32 getRandomNumber(uint32 limit) {

 uint32 random_number = 0;

 os_.aes().ccm_mac(  16, entropy_accumulator,
                     16, entropy_accumulator,
                     NODE_PRIVATE_KEY, counter++);

 memcpy(&random_number , entropy_accumulator, 4);

 return (random_number % limit);
}
```

Abbildung 5.7.: Implementierung des sicheren PRNG mittels AES-CCM

und weit verbreitete Algorithmen. So schlagen die Autoren aus [124] vor, den Energieverbrauch in drahtlosen Kommunikationsnetzen zu reduzieren, indem der Nachrichteninhalt komprimiert wird. Die Autoren argumentieren, dass die Energie, die zum Senden eines Bit benötigt wird, um den Faktor 1000 höher ausfällt, als die Energie, die eine 32-Bit Berechnung benötigt. Sie vergleichen hierfür mehrere Kompressionsverfahren miteinander, darunter LZO, ZLib, ncompress, bzip2 und PPMd. Durch die Anpassung dieser Algorithmen kann auf

diese Weise eine Energieersparnis von 43 % bis 69 % erreicht werden.

In der Arbeit von [125] werden diese Algorithmen hinsichtlich ihrer Eignung für drahtlose Sensornetze untersucht. Ihre Untersuchungen ergeben hierbei, dass der Speicherverbrauch dieser Algorithmen zu hoch für die Beschränkungen eines Sensorknotens wie dem MSP430 ist und stellen eine optimierte Version des LZW-Algorithmus vor, den sie S-LZW nennen. S-LZW ist eine Weiterentwicklung bereits existierender embedded Versionen [114, 126] des LZW-Algorithmus, der 1984 von den Lempel, Ziv und Welch entwickelt wurde und unter anderem vom Bildformat GIF [127] verwendet wird. Die Autoren von [125] geben an, dass S-LZW bei der Kompression von Sensordaten Kompressionsraten zwischen 1,11 und 4,58 erreicht, wodurch der Stromverbrauch eines drahtlosen Sensornetzes durchschnittlich um den Faktor 1,5 bis 2,5 verringert werden kann. Es existiert eine Implementierung von S-LZW für den MSP430[2], die vom Autor dieser Arbeit für die iSense-Plattform angepasst und in das DPS-Protokoll integriert wurde.

5.5. Implementierung von Stubs und Skeletons

Das DPS-Protokoll dient dazu, die Implementierung einer Schicht eines Protokollstapels mittels RPC-Mechanismen mit benachbarten Sensorknoten zu teilen. Das Konzept und die Implementierung dieses Protokolls wurden in den vorangegangenen Abschnitten beschrieben. Zum Betrieb des DPS-Protokolls ist es zusätzlich jedoch notwendig, den entsprechenden Client-Stub und Server-Skeleton für eine oder mehrere Schichten des Protokollstapels zu implementieren. An diesem Punkt muss die Entscheidung gefällt werden, welche Schichten des Stacks auf dem Client implementiert werden sollen und wie die zugehörigen Schnittstellen im Client-Stub und Server-Skeleton aufgebaut sind. Diese Fragestellung wird im Folgenden beantwortet. Bei der Beantwortung dieser Frage ergeben sich zwei Herangehensweisen - entweder konzeptionell über die Funktion der einzelnen Schichten im Stack oder pragmatisch über die Programmgröße und den Implementierungsaufwand. In diesem Abschnitt sollen beide Herangehensweisen untersucht werden.

Im Anschluss an diesen Abschnitt werden zwei Beispiele für solche Implementierungen vorgestellt: Zunächst wird in Abschnitt 5.6 die Implementierung des Stubs und Skeletons auf der Vermittlungsschicht vorgestellt. Anschließend wird in Abschnitt 5.7 die Implementierung eines Routing-Protokolls vorgestellt, wobei hierbei zusätzlich zu der Implementierung auf den Sensorknoten eine zentralisierte Komponente verwendet wird.

5.5.1. Konzeptionelle Vorgehensweise

Konzeptionell kann das DPS-Protokoll auf jeder Schicht des Protokollstapels implementiert werden. Zur Wahl stehen also entweder die Anwendungs-, Transport-,

[2] https://sites.google.com/site/cmsadler/

Vermittlungs- oder die Adaptionsschicht. Die Sicherungs- und Bitübertragungsschicht sind bei Sensorknoten häufig direkt in Hardware implementiert und werden darüber hinaus vom DPS-Protokoll zum Nachrichtenaustausch verwendet, weshalb diese im Rahmen dieses Abschnitts nicht näher untersucht werden.

Im Folgenden wird der Einsatz des DPS-Protokoll auf den vier verbleibenden Schichten kurz beschrieben und die Parameter der entsprechenden Stubs und Skeletons skizziert. Die Liste ist jedoch nicht vollständig und dient lediglich dazu, einen Überblick über die entsprechende Methode zu erlangen:

Anwendungsschicht

Der Einsatz des DPS-Protokolls auf der Anwendungsschicht sorgt dafür, dass der DPS-Client keine einzige Schicht des Protokollstapels implementiert und stattdessen nur die Anwendungslogik umsetzt, die dann ein oder mehrere Protokolle der Anwendungsschicht verwendet. In Abhängigkeit vom gewählten Einsatzszenario müssen deshalb unter Umständen mehrere Stubs und Skeletons implementiert werden, z.B. für das CoAP-Protokoll [41]. Im Gegensatz zu der sehr geringen Anzahl von Schnittstellen eines Transport- oder Vermittlungsschichtprotokolls (siehe unten), bieten die Protokolle auf der Anwendungsschicht eine Vielzahl von Schnittstellen. Hierdurch steigt der Implementierungsaufwand.

Transportschicht

In diesem Fall müssen die Stubs und Skeletons für jedes verwendete Transportschichtprotokoll implementiert werden. In einer Sensornetzanwendung sind dies typischerweise UDP und ICMP[3]. Sollen darüber hinaus noch weitere Transportprotokolle, wie z.B. TCP, eingesetzt werden, müssen auch hierfür die Stubs und Skeletons bereitgestellt werden. Bei einem Einsatz des DPS-Protokolls auf der Transportschicht bietet der entsprechende Stub z.B. einen UDP-Socket für die Anwendung an, der die folgenden Parameter erwartet: Quell-Port, Ziel-Port, IP-Adresse des Ziels, Länge der zu sendenden Daten und die zu sendenden Daten. Der TCP-Stub erhält die entsprechenden Parameter. ICMP hingegen erhält lediglich die IP-Adresse des Ziels und zusätzlich den Typ und Code der zu sendenden ICMP-Nachricht, z.B. Typ = 1 (Destination Unreachable) und Code = 0 (No route to destination).

Zusätzlich muss auch für den Empfang von Nachrichten eine Schnittstelle bereitgestellt werden und der entsprechende Server-Skeleton implementiert werden. Da das DPS-Protokoll lediglich die Reihenfolgeerhaltung und Nachrichtenverlustbehandlung implementiert, im Gegensatz zu TCP jedoch keine Mechanismen zur Fluss- oder Staukontrolle beinhaltet, kann dies zu Einschränkungen beim Einsatz von TCP über das DPS-Protokoll führen.

[3] In diesem Kontext wird ICMP als Transportprotokoll betrachtet, obwohl es eher der Vermittlungsschicht zuzuordnen ist.

Vermittlungsschicht

Wenn das DPS-Protokoll auf der Vermittlungsschicht eingesetzt wird, muss lediglich ein Stub und Skeleton für die IPv6-Schicht implementiert werden, da diese als sogenannte Narrow Waist fungiert. Der Begriff Narrow Waist (engl. für „schmale Taille“) bezieht sich in diesem Fall auf die Eigenschaft von IP, in einer sehr großen Anzahl von unterschiedlichen Netzwerken eingesetzt zu werden und sowohl oberhalb als auch unterhalb eine Vielzahl unterschiedlicher Protokolle miteinander zu verbinden (vgl. Abbildung 2.6 auf Seite 24). Der Einsatz des DPS-Protokolls auf der Vermittlungsschicht bietet sich deshalb in vielen Fällen an, da unabhängig von den auf dem Client implementierten Transportprotokollen eine gemeinsame Schnittstelle verwendet werden kann. Der IPv6-Stub muss hierbei eine Sende- und Empfangsfunktion implementieren, welche die folgenden Parameter verwendet: IP-Adresse des Ziels, die Länge der zu sendenden Daten, die zu sendenden Daten sowie das Hop-Limit. Die Traffic-Class und das Flow-Label hingegen werden nur in Ausnahmefällen benötigt, da diese von der 6LoWPAN-Adaptationschicht im Rahmen der Kompression der Nachricht entfernt werden.

Adaptionsschicht

Falls das DPS-Protokoll auf der 6LoWPAN-Adaptionsschicht implementiert wird, muss der DPS-Client sowohl die IPv6 als auch die Transportschichtprotokolle selbst implementieren. Der zugehörige Client-Stub sitzt ähnlich wie auf der Vermittlungsschicht als Narrow Waist unterhalb der restlichen Protokolle und oberhalb der Medienzugriffsschicht. Falls in dem Sensornetz ein Routing auf IP-Ebene verwendet wird, so muss dieses jedoch auch auf dem Client implementiert werden. Der Stub und Skeleton implementiert jeweils eine Sende- und Empfangsschnittstelle, die als Parameter eine unkomprimierte IPv6-Nachricht mit allen darin enthaltenen Feldern plus der Länge der zu sendenden Daten und der zu sendenden Daten sowie der MAC-Adresse des Empfängers.

5.5.2. Pragmatische Vorgehensweise

Neben der konzeptionellen Vorgehensweise kann die Wahl der Schicht für den Einsatz des DPS-Protokolls auch pragmatisch anhand der Quellcodegrößen der entsprechenden Schichten geschehen. Als Beispiel für eine IPv6-Implementierung für Sensornetze wird in diesem Abschnitt der IPv6-Stack des iSense-Betriebssystems verwendet, das sowohl die JN5139 als auch die JN5148-Hardwareplattform unterstützt. Wie bereits im Rahmen von Abschnitt 1.1 beschrieben, ist dieser IPv6-Stack jedoch nur auf der JN5148-Plattform lauffähig, da die Programmgröße des IPv6-Stack zusammen mit dem Betriebssystem bereits den verfügbaren Programmspeicher der JN5139-Plattform übersteigt. Dies war der ursprüngliche Grund für die Entwicklung des Konzepts hinter dieser Arbeit. Es ist jedoch möglich, das DPS-Protokoll zusammen mit einem Teil des IPv6-Stack auf dieser

Tabelle 5.1.: Quellcodegröße der einzelnen Schichten unterschiedlicher Implementierungen des IPv6-Stack

Protokoll	iSense (JN5139)		Contiki (JN5139)		Contiki (MSP430)	
UDP	1,9 KB	3,8 %	0,3 KB	0,9 %	0,7 KB	3,4 %
ICMP	1,8 KB	3,6 %	1,3 KB	3,7 %	0,8 KB	3,9 %
IPv6	31,6 KB	62,9 %	12,3 KB	34,9 %	7,4 KB	36,5 %
ND	7,6 KB	15,1 %	13,3 KB	37,8 %	6,8 KB	33,5 %
6LoWPAN	7,3 KB	14,5 %	8,0 KB	22,7 %	4,6 KB	22,7 %
$\sum$	50,2 KB	100 %	35,2 KB	100 %	20,3 KB	100 %

Hardwareplattform auszuführen. Das Ziel des DPS-Protokolls ist es folglich, die Quellcodegröße des IP-Stacks auf dieser Plattform so weit zu reduzieren, dass der Betrieb einer Anwendung zusammen mit einem Teil des IP-Stack und dem DPS-Protokoll möglich ist. Die Vorgehensweise kann hierbei analog auch auf andere Plattformen übertragen werden.

Um zu entscheiden, auf welcher Schicht die Stubs und Skeletons des DPS-Protokolls implementiert werden sollen, muss zunächst die Größe der einzelnen Schichten analysiert werden. Hierfür wurden die Quellcodegrößen der einzelnen Schichten des iSense- und Contiki-IPv6-Stack auf der JN5139- und MSP430-Plattform ermittelt, die in Tabelle 5.1 zusammengefasst sind. Aus den in der Tabelle dargestellten Quellcodegrößen geht hervor, dass die beiden Protokolle UDP und ICMP nur einen sehr kleinen Teil der Gesamtgröße des IPv6-Stacks ausmachen (4,6 % bis 7,4 %), während vor allem das IPv6-Protokoll mehr als ein Drittel der Quellcodegröße ausmacht. Wie bereits im Rahmen der konzeptionellen Vorgehensweise erläutert wurde, können die Stubs und Skeletons des DPS-Protokolls für jedes der in Tabelle 5.1 dargestellten Protokolle implementiert werden, wodurch sich der Speicherplatzbedarf des entsprechenden Protokolls verringert, während alle darunter liegenden Schichten nur auf dem DPS-Server implementiert werden (vgl. Abbildung 3.2(b) auf Seite 32).

Es ist folglich von Vorteil für den Client, die Stubs und Skeletons auf einer möglichst hohen Protokollschicht zu implementieren, um die Reduktion der Quellcodegröße zu maximieren. Da das IPv6-Protokoll den größten Teil der Quellcodegröße ausmacht, sollten die Stubs und Skeletons entweder auf dieser Schicht oder auf einer der darüber liegenden Schichten implementiert werden. Auf diese Weise kann die Quellcodegröße für den Client um insgesamt 33,6 kB bis 46,5 kB verringert werden (18,8 kB auf dem MSP430). Dies wird im nachfolgenden Abschnitt genauer beschrieben.

5.6. Beispiel 1: Vermittlungsschicht

Um die Funktionalität des DPS-Protokolls zu testen, wurden der Client-Stub und der zugehörige Server-Skeleton für das IPv6-Protokoll des iSense-IPv6-Stacks

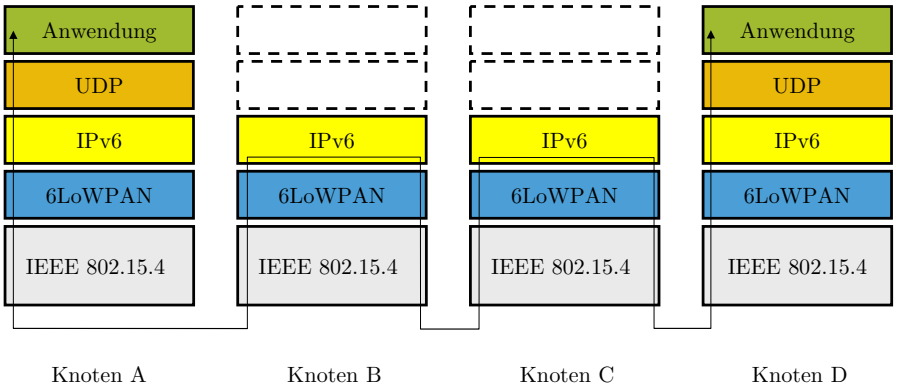


Abbildung 5.8.: Austausch von UDP-Nachrichten mittels des IPv6-Protokolls

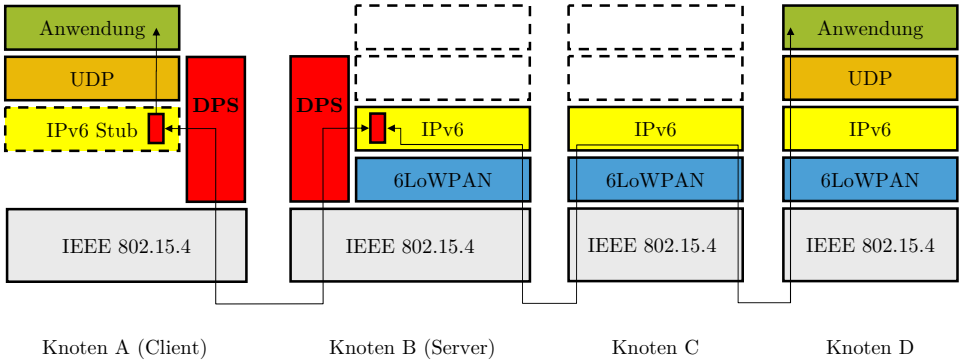


Abbildung 5.9.: Austausch von UDP-Nachrichten mittels des DPS-Protokolls unter Verwendung des IPv6-Stubs und -Skeletons

implementiert. Dieser IP-Stack ist Bestandteil des iSense-Betriebssystems der Firma coalesenses und unterstützt die beiden Hardwareplattformen JN5139 sowie JN5148, die im WISEBED-Testbed vorhanden sind und für die Evaluation in Kapitel 6 verwendet werden. Die in Abschnitt 5.5 vorgestellten Vorgehensweisen wurden als Grundlage für die Wahl der zur Demonstration des DPS-Protokolls zu implementierenden Stubs und Skeletons verwendet.

Das IPv6-Protokoll macht den größten Teil der Quellcodegröße des IPv6-Stacks aus und wurde deshalb für die Implementierung des Client-Stubs und Server-Skeletons ausgewählt. Dies wird in diesem Abschnitt näher erläutert. Durch die Wahl des IPv6-Protokolls reduziert sich die Quellcodegröße des IPv6-Stack auf dem Client theoretisch um 46,5 KB (92,6 %), wobei jedoch noch die Größe des DPS-Protokolls sowie des Client-Stubs berücksichtigt werden muss. Dies wird in Abschnitt 6.3 im Rahmen der Evaluation genauer untersucht.

Im Folgenden wird der Begriff Server für einen Sensorknoten vom Typ JN5148 verwendet, der alle Schichten des Protokollstapels implementiert, während der Begriff Client für einen Sensorknoten vom Typ JN5139 verwendet wird, der nur eine Teilmenge der Schichten implementiert.

5.6.1. Vorgehensweise

Dieser Abschnitt veranschaulicht die Auswirkungen dieser Vorgehensweise auf den IPv6-Stack und die Kommunikation im Sensornetz. Abbildung 5.8 zeigt, wie zwei Anwendungen auf unterschiedlichen Sensorknoten (Knoten A und Knoten D) Informationen über das Netzwerk austauschen. Hierbei werden die UDP-Pakete von Knoten A mittels des IPv6-Protokolls verschickt, von der 6LoWPAN-Schicht komprimiert und über die 802.15.4-Sicherungs- und Bitübertragungsschicht versendet. Auf den zwischen dem Start- und Zielknoten liegenden Knoten B und C wird das Paket von der IPv6-Schicht empfangen und an die durch das Routing-Protokoll ermittelte nächste Station weitergeleitet. Auf der Seite des Empfängers (Knoten D) durchwandert die Nachricht wieder sämtliche Schichten des TCP/IP-Referenzmodells und wird an die Anwendung ausgeliefert.

Wird hingegen, wie in Abbildung 5.9 dargestellt, das DPS-Protokoll auf der Vermittlungsschicht verwendet, so wird auf dem Client beim Versand von UDP-Paketen anstelle der 6LoWPAN-Schicht stattdessen das DPS-Protokoll verwendet. Hierfür implementiert der Server (Knoten B in Abbildung 5.9) den Server-Skeleton in der IPv6-Schicht und das DPS-Protokoll zusätzlich zu den restlichen Komponenten des IPv6-Stacks, während der Client (Knoten A in Abbildung 5.9) den IPv6-Stub sowie das DPS-Protokoll anstelle des IPv6-, 6LoWPAN- und Routing-Protokolls implementiert. Nachdem der Server-Knoten B die Nachricht empfangen hat, entspricht die Vorgehensweise dem in Abbildung 5.8 dargestellten Ablauf. Auf diese Weise wird zwar einerseits die Quellcodegröße für den Client reduziert, gleichzeitig kommt es jedoch zu zwei Einschränkungen:

1. Durch den Wegfall der 6LoWPAN-Adaptionsschicht müssen die IPv6-Nachrichten unkomprimiert zwischen dem Server und Client ausgetauscht werden.
2. Durch den Wegfall des Routing-Protokolls auf dem Client müssen sich der DPS-Client und Server in Funkreichweite zueinander befinden.

5.6.2. Umsetzung

Die Auswirkungen dieser Einschränkungen werden im nachfolgenden Kapitel 6 im Rahmen der Evaluation genauer untersucht. An dieser Stelle wird zunächst noch auf einige Implementierungsdetails eingegangen: Abbildung 5.10 zeigt ein UML-Klassendiagramm des IPv6-Stubs und -Skeletons. Sowohl der Stub als auch der Skeleton implementieren das Interface DpsEventHandler, über das beide Ereignisse vom DPS-Protokoll erhalten (siehe Abschnitt 5.1.3). Darüber hinaus besitzen beide Klassen Referenzen auf das iSense-Betriebssystem, das UDP- und ICMP-Protokoll sowie eine `input()` und eine `output()` Funktion. Diese beiden Funktionen werden für den Versand und den Empfang von IPv6-Nachrichten genutzt und bilden folglich eine der Schnittstellen des DPS-Protokolls. Die restlichen Funktionen der beiden Klassen werden im Folgenden für den Stub und Skeleton kurz beschrieben. Es wird hierbei das Verhalten nach dem Aufbau

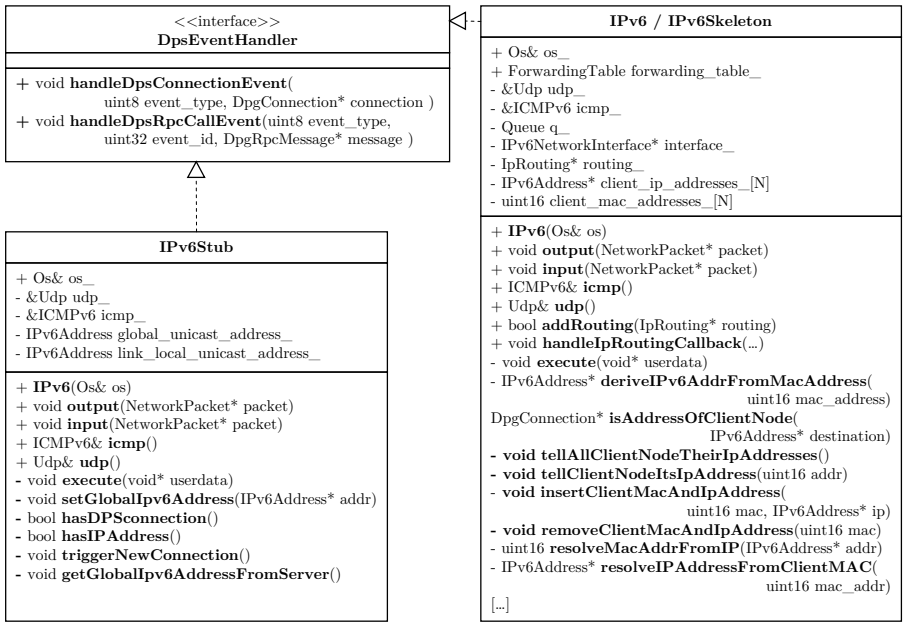


Abbildung 5.10.: Klassendiagramm des Interfaces DpsEventHandler und der Klassen IPv6Stub und IPv6Skeleton (gekürzt)

der DPS-Verbindung beschrieben:

Nachdem der IPv6-Stub eine DPS-Verbindung zu einem Skeleton aufgebaut hat, erhält er eine IPv6-Adresse vom Server, indem der Server die Funktion `setGlobalIpv6Address()` aufruft. Er kann seine eigene IPv6-Adresse jedoch auch mittels `getGlobalIpv6AddressFromServer()` vom Server erfragen. Beide Funktionen verwenden die Bestätigungen des DPS-Protokolls. Die `output()` Funktion wird vom Stub zum Senden von IPv6-Nachrichten verwendet. Hierfür serialisiert die `output()` Funktion das IPv6-Paket und sendet es an den Skeleton. Umgekehrt werden eintreffende Pakete über die Funktion `handleDpsRpcCallEvent()` empfangen, deserialisiert und anschließend in der Funktion `input()` vom Stub verarbeitet. Sowohl die `input()` als auch die `output()` Funktion verwenden keine Bestätigungen.

Auf der Seite des IPv6-Skeletons wird nach dem erfolgreichen Aufbau der DPS-Verbindung zunächst die IPv6-Adresse des Clients auf Basis seiner MAC-Adresse ermittelt. Dies kann entweder mittels der IPv6-Stateless-Address-Autoconfiguration [128] passieren (`deriveIPv6AddrFromMacAddress()`) oder z.B. unter Verwendung von DHCPv6 [129]. Nachdem die IPv6-Adresse des Clients ermittelt wurde, wird diese zunächst zusammen mit der MAC-Adresse des Clients in den Datenstrukturen `client_mac_addresses_[]` und `client_ip_addresses_[]` gespeichert und anschließend an den Client durch den Aufruf der Funktion `tellClientNodeItsIpAddress()` weitergeleitet. Diese Datenstrukturen werden für das Weiterleiten von eintreffenden IPv6-Paketen verwendet, die für einen der angeschlossenen Clients bestimmt sind. Empfängt ein DPS-Server eine IPv6-Nachricht, so wird die Zieladresse dieses Pakets mit den in der Datenstruktur `client_ip_addresses_[]` gespeicherten Adressen verglichen. Beinhaltet diese

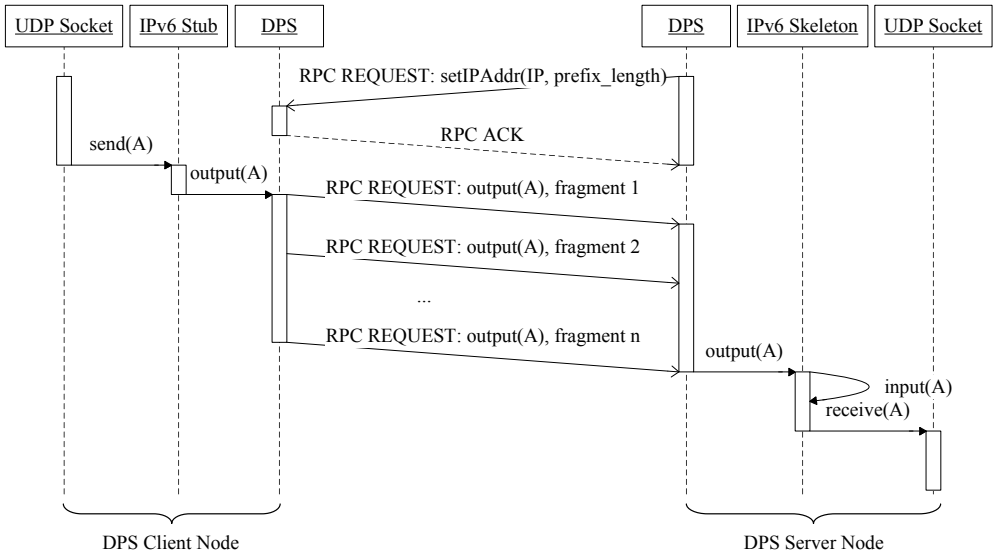


Abbildung 5.11.: Sequenzdiagramm beim Versenden einer UDP-Nachricht vom IPv6-Stub an den IPv6-Skeleton. In diesem Beispiel kommunizieren die Anwendung auf dem Client und die Anwendung auf dem Server über UDP direkt miteinander

Datenstruktur die Zieladresse, so wird die Nachricht mittels des DPS-Protokolls an den entsprechenden Client weitergeleitet. Andernfalls wird die Nachricht mittels des IP-Protokolls weitergeleitet. Das Versenden und Empfangen von Nachrichten an/von dem Client geschieht hierbei Analog zum IPv6-Stub (siehe oben). Bei ausgehenden Nachrichten, die demzufolge vom DPS-Protokoll an die `output()` Funktion des Skeleton weitergeleitet werden, muss zusätzlich zum Weiterleiten des Pakets mittels des IPv6-Protokolls bei Bedarf eine Route zum angegebenen Ziel aufgebaut werden.

5.6.3. Beispiel

Ein Beispiel für das Versenden eines UDP-Pakets vom IPv6-Stub an den Skeleton ist in Abbildung 5.11 als Sequenzdiagramm dargestellt. Um die Darstellung zu vereinfachen, nehmen in diesem Beispiel nur zwei Knoten an der Kommunikation teil, d.h. die Anwendung auf dem Client sendet ihre Nachrichten an die Anwendung auf dem Server.

Nachdem die DPS-Verbindung aufgebaut wurde, teilt der Skeleton dem Stub seine IPv6-Adresse mit. Der Stub antwortet mit einer Bestätigung, nachdem der zugehörige RPC-Request empfangen wurde. Anschließend sendet der Client über einen UDP-Socket ein Paket an den Server. Hierfür wird die `send()` Funktion des UDP-Protokolls verwendet, die das UDP-Paket an den IPv6-Stub übergibt. Hier wird das Paket innerhalb der `output()` Funktion serialisiert und an das DPS-Protokoll weitergegeben. Das DPS-Protokoll sendet die serialisierte Nachricht anschließend als RPC-Request an das DPS-Protokoll auf dem Server, wobei der

Fragmentierungsmechanismus das DPS-Protokolls die Nachricht in insgesamt n Fragmente aufteilt. Diese werden ohne Bestätigungen versendet und nach dem erfolgreichen Empfang des letzten Fragments vom DPS-Protokoll auf dem Server wieder zusammengesetzt. Das zusammengesetzte Paket wird deserialisiert und an die `output()` Funktion des IPv6-Skeleton übergeben. Da die Zieladresse des IPv6-Pakets in diesem Beispiel der Adresse des Servers entspricht, wird die `input()` Funktion des Skeleton aufgerufen, welche die Nachricht an den UDP-Socket weitergibt.

5.7. Beispiel 2: Routing-Protokoll

Das DPS-Protokoll dient dazu, einem Sensorknoten die Funktionalität eines Protokolls zugänglich zu machen, ohne das dieser Sensorknoten das Protokoll selbst implementieren muss. Im vorangegangenen Abschnitt wurde beschrieben, wie ein Sensorknoten Nachrichten über IPv6 versenden kann, ohne selbst das IPv6-Protokoll zu implementieren und stattdessen die Implementierung eines benachbarten Sensorknotens über RPC-Mechanismen mittels des DPS-Protokolls zu verwenden.

Das DPS-Protokoll ist jedoch nicht darauf beschränkt, Implementierungen von Protokollen zwischen benachbarten Sensorknoten zu teilen. Stattdessen kann z.B. auch die Implementierung des Protokolls außerhalb des Sensornetzes stattfinden. Dies wird in diesem Abschnitt am Beispiel eines zentralisieren Routing-Protokolls erläutert, das über die Infrastruktur des WISEBED-Testbed mit jedem Sensorknoten im Netz direkt verbunden ist.

5.7.1. Existierende Routing-Protokolle des iSense-IPv6-Stack

Für den IPv6-Stack der iSense-Plattform existieren zum Zeitpunkt dieser Arbeit mehrere Routing-Protokolle. So wurden im Rahmen der Diplomarbeit von Christian Tille [49], die vom Autor dieser Arbeit betreut wurde, die beiden Routing-Protokolle DYMO [50] und DYMO-low [51] implementiert. Darüber hinaus existieren noch die beiden statischen Routing-Protokolle TrivialRouting und StaticRouting. Diese Routing-Protokolle werden im Folgenden beschrieben.

DYMO ist ein von der IETF entwickeltes Distanzvektor-Routing-Protokoll, das eine Weiterentwicklung des Routing-Protokolls AODV darstellt (siehe RFC 3561 [130]). DYMO ist ein Route-Over-Routing-Protokoll, d.h. das Routing wird oberhalb der IP-Schicht durchgeführt, so dass das zu verschickende IP-Paket auf jedem Sensorknoten entlang der Route wieder zusammengesetzt wird und somit jeder dieser Sensorknoten einem IP-Hop entspricht. Alternativ hierzu existiert eine Implementierung des Routing-Protokolls DYMO-low. Hierbei handelt es sich um eine Abwandlung von DYMO, die als Mesh-Under-Routing-Protokoll arbeitet, d.h. das Routing wird unterhalb der IP-Schicht auf der 6LoWPAN-Schicht durchgeführt. Dies bedeutet, dass das IP-Paket u.U. erst beim Ziel-Knoten zusammengesetzt wird und die gesamte Route im Sensornetz

als ein einziger IP-Hop erscheint. Die beiden Routing-Protokolle DYMO und DYMO-low wurden in der Arbeit von [49] implementiert und in kleinen Netzwerken mit bis zu sechs Sensorknoten getestet. Tests am Institut für Telematik auf dem WISEBED-Testbed haben jedoch gezeigt, dass beide Implementierungen in größeren Sensornetzen nicht mehr oder nur sehr unzuverlässig arbeiten. So war der Nachrichtenaustausch mit beiden Protokollen nur über eine maximale Entfernung von zwei Hops möglich.

Alternativ zu diesen beiden dynamischen Routing-Protokollen, die zur Laufzeit Routen im Sensornetz auf Basis der aktuellen Netzwerktopologie erstellen, können auch zwei statische Routing-Protokolle eingesetzt werden. Das erste dieser beiden Protokolle ist das sogenannte TrivialRouting, das mit link-lokalen IPv6-Adressen arbeitet und ein Single-Hop-Routing-Protokoll darstellt, d.h. der nächste IP-Hop von A nach B entspricht immer B. Das TrivialRouting eignet sich also nur für sehr kleine Sensornetze, bei denen sich der Empfänger eines IP-Pakets immer in unmittelbarer Funkreichweite des Senders befindet. Andernfalls muss das TrivialRouting mit einem Mesh-Under-Routing, wie z.B. DYMO-low, kombiniert werden. Das StaticRouting hingegen ermöglicht es dem Entwickler, dem Sensornetz programmatisch eine feste Topologie zuzuweisen, die dann während der Laufzeit für alle IP-Pakete verwendet wird. Hierdurch sind auch größere Sensornetze möglich, die jedoch auf die Verwendung einer statischen Topologie angewiesen sind. Da sich die Links in einem drahtlosen Netzwerk jedoch ständig ändern können, eignet sich dieses Protokoll nur sehr eingeschränkt für den Betrieb eines Sensornetzes über einen längeren Zeitraum.

Da die iSense-Plattform zum Zeitpunkt dieser Arbeit kein geeignetes und funktionsfähiges Routing-Protokoll für den Betrieb eines großen Sensornetzes (wie z.B. im WISEBED-Testbed mit > 40 Sensorknoten) zur Verfügung stellt, wurde das DPS-Protokoll verwendet, um mittels einer zentralisierten Komponente über die Infrastruktur des Testbeds ein Routing-Protokoll umzusetzen, das den Betrieb und die Evaluation eines IPv6-Netzes im WISEBED-Testbed ermöglicht. Dieses Protokoll wird im nächsten Abschnitt näher beschrieben.

5.7.2. Zentralisiertes Routing-Protokoll

Das zentralisierte Routing-Protokoll (ZRP) wurde als Java Anwendung implementiert, die über die Infrastruktur des WISEBED-Testbeds mit den Sensorknoten verbunden ist und auf diese Weise mit den Routing-Stubs bidirektional kommunizieren kann. Dies ist in Abbildung 5.12 dargestellt: Alle Sensorknoten im Testbed sind via USB mit lokalen Gateways verbunden, die mit dem Portal-Server über LAN verbunden sind. Der Portal-Server stellt eine API zur Kommunikation mit den Sensorknoten zur Verfügung, die es erlaubt, jeden einzelnen Sensorknoten individuell zu programmieren, neu zu starten und Nachrichten mit diesem Sensorknoten auszutauschen. Diese API wird vom Routing-Skeleton des ZRP dazu verwendet, um über das DPS-Protokoll mit den Routing-Stubs auf den Sensorknoten zu kommunizieren. Der Routing-Stub wurde auf der IPv6-Schicht der Sensorknoten als Route-Over-Routing-Protokoll implementiert (siehe

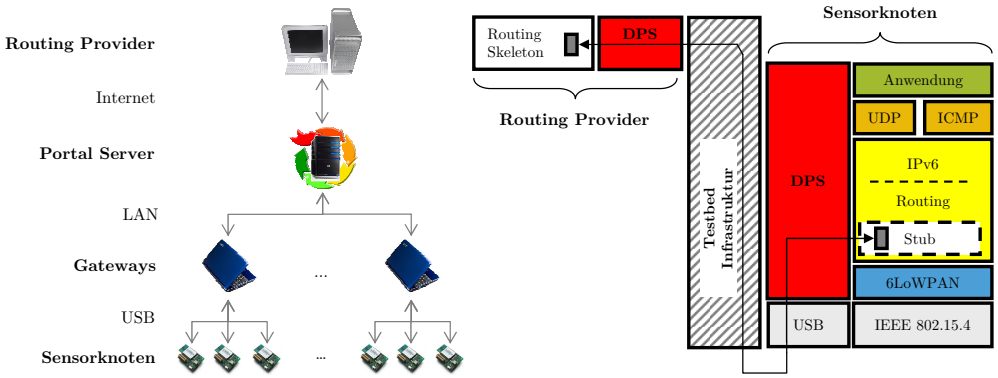


Abbildung 5.12.: Kommunikation zwischen den Komponenten der Infrastruktur des WISEBED-Testbeds (links) sowie zwischen dem Routing-Skeleton des ZRP und den Sensorknoten im Testbed (rechts)

Abschnitt 5.7.1). Das ZRP besteht hierbei aus drei Funktionskomponenten:

Die erste Komponente ist für das Sammeln von Topologieinformationen zuständig. Diese Komponente dient dazu, dem ZRP jederzeit den aktuellen Zustand der Links im Sensornetzwerk mitzuteilen. Die zweite Komponente kann aus den gesammelten Topologieinformationen einen Graphen des Netzwerk konstruieren und in diesem Graphen den kürzesten Weg zwischen zwei beliebigen Sensorknoten berechnet. Die dritte Komponente bildet die Schnittstelle zum Routing-Stub auf den Sensorknoten, ist also für den Aufbau der Routen zwischen den Sensorknoten zuständig. Diese Komponenten werden in den folgenden Abschnitten genauer beschrieben.

5.7.3. Sammeln von Topologieinformationen

Die Aufgabe eines dynamischen Routing-Protokolls ist es, eine Route zwischen zwei Teilnehmern des Netzes zu finden. Diese Route wird basierend auf der aktuellen Topologie des Netzwerkes berechnet, d.h. es muss berücksichtigt werden, welche Sensorknoten miteinander kommunizieren können und wie die Qualität dieser Verbindung ist. Eine Verbindung wird in diesem Zusammenhang auch als Link bezeichnet. Die Topologieinformationen werden vom ZRP gesammelt, indem es in regelmäßigen Abständen Beacon-Nachrichten von allen Sensorknoten im Netzwerk verschicken lässt (z.B. jede Sekunde).

Beacon-Nachrichten beinhalten eine Beacon-Id, die zur Identifikation des Beacons verwendet wird. Sobald ein Sensorknoten ein Beacon von einem benachbarten Knoten erhält, teilt er dies dem ZRP mit, wobei die folgenden Informationen übertragen werden: der Empfangszeitpunkt, die eigene MAC-Adresse, die MAC-Adresse des Absenders des Beacons, die Beacon-Id sowie die Signalstärke (RSSI) und Signalqualität (LQI) des empfangenen Beacons. Diese Beacon-Meldungen werden vom ZRP für jeden Sensorknoten gespeichert, wobei für jedes Knotenpaar und Richtung (z.B. A-B und B-A) die letzten k Meldungen gespeichert werden.

Über die gespeicherten Beacon-Meldungen kann das ZRP bestimmen, welche Knoten miteinander kommunizieren können und wie die Qualität dieser Verbindung beschaffen ist. Hierbei unterstützt das ZRP insgesamt drei Metriken: Die Signalstärke, die Signalqualität und die Paketankunftsrate: Bei der Signalstärke und Signalqualität wird der Durchschnitt der letzten k Beacon-Meldungen verwendet, während bei der Paketankunftsrate die sogenannte ETX-Metrik [131] verwendet wird. Die ETX-Metrik wird unter anderem vom Routing-Protokoll CTP (Collection-Tree-Protocol [132]) eingesetzt und stellt eine der beiden Standard-Metriken des RPL-Routing-Protokolls dar [133]. Als solche wird es unter anderem in den RPL-Implementierungen des TinyOS- und des Contiki-IPv6-Stacks verwendet [134]. Die Abkürzung ETX sieht hierbei für *Expected-Transmission-Count* und gibt an, wie oft eine Nachricht voraussichtlich über einen Link gesendet werden muss, um am Ziel anzukommen.

Die ETX-Metrik wird vom ZRP über die letzten k Beacon-Nachrichten wie folgt berechnet: Das ZRP weiß, wie viele Beacons von jedem Sensorknoten in einem gegebenen Zeitintervall gesendet werden (z.B. zehn Nachrichten in zehn Sekunden). Für jeden Link ermittelt das ZRP nun die Anzahl der tatsächlich empfangenen Beacon-Nachrichten. Hieraus wird die Paketankunftsrate in beiden Richtungen des Links ermittelt - wurden z.B. nur acht Nachrichten empfangen, so beträgt die Paketankunftsrate $8/10 = 0{,}8$. Die ETX-Metrik des Links zwischen A und B ergibt sich anschließend aus Gleichung 5.5, wobei d_f die *Forward-Delivery-Ratio* angibt, also die Paketankunftsrate von A nach B, während d_r die *Reverse-Delivery-Ratio*, also die Paketankunftsrate von B nach A angibt.

$$ETX = \frac{1}{d_f * d_r} \tag{5.5}$$

Die ETX Metrik beträgt somit im besten Fall 1,0 (100 % Paketankunftsrate in beiden Richtungen) und steigt für schlechtere Paketankunftsraten. Beispiel: Knoten A und B senden beide zehn Beacon-Nachrichten in zehn Sekunden. Knoten A erhält sieben Beacon-Nachrichten von B, während B nur drei Beacon-Nachrichten von A erhält. Hieraus ergeben sich $d_f = 7/10 = 0{,}7$ und $d_r = 3/10 = 0{,}3$ und somit $ETX = \frac{1}{0{,}7*0{,}3} \approx 4{,}76$.

Da es sich bei dem ZRP um ein Route-Over-Routing-Protokoll handelt, nehmen nur diejenigen Sensorknoten am Routing-Protokoll teil, die auch das IPv6-Protokoll implementieren. Im vorliegenden Fall sind dies nur die DPS-Server, die den IPv6-Skeleton implementieren. Die DPS-Clients, die lediglich über den IPv6-Stub verfügen, nehmen nicht am Routing-Protokoll teil. Beacon-Nachrichten werden folglich nur von den DPS-Servern verschickt, und nur die DPS-Server benachrichtigen das ZRP über den Empfang von Beacon-Nachrichten. Die Clients sind automatisch über den Server zu erreichen, zu dem sie eine DPS-Verbindung aufgebaut haben. Hierfür besitzt das ZRP eine separate Schnittstelle, über welche die Server das ZRP beim Zustandekommen einer DPS-Verbindung informieren. Der entsprechende Link wird mit der bestmöglichen Metrik in die Topologieinformationen aufgenommen. Im Fall der ETX-Metrik erhalten die Links zwischen Client und Server einen Wert von 1,0.

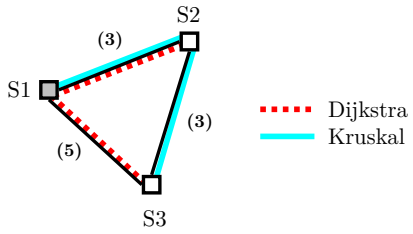


Abbildung 5.13.: Kürzeste Pfade aller Knoten im Graphen zu S1 nach Dijkstra und Minimaler Spannbaum ausgehend von S1 nach Kruskal

5.7.4. Berechnen von Routen im Netzwerk

Die gesammelten Links des Netzwerkes werden nun zu einem Graphen zusammengesetzt. Der Graph repräsentiert hierbei die Verbindungen im Netzwerk und wird in Form einer Adjazenzliste gespeichert. Auf der Grundlage dieser Adjazenzliste kann nun der kürzeste Weg zwischen zwei beliebigen Knoten im Graphen berechnet werden. Hierfür kann das ZRP zwei verschiedene Algorithmen einsetzen - die Algorithmen von Dijkstra und Kruskal. Der Algorithmus von Dijkstra berechnet ausgehend von einem Knoten den kürzesten Pfad zu allen anderen Knoten im Netzwerk, wobei immer garantiert ist, dass er den kürzesten Pfad findet. Es wird folglich ein Spannbaum[4] des Graphen berechnet, jedoch nicht der minimalen Spannbaum[5]. Aus diesem Grund kann alternativ der Algorithmus von Kruskal verwendet werden, der den minimalen Spannbaum eines Graphen berechnet.

Um den Unterschied zwischen beiden Algorithmen zu verdeutlichen, ist in Abbildung 5.13 ein Beispiel für einen Graphen mit drei Knoten gegeben. Der Graph in Abbildung 5.13 besteht aus den Knoten S1, S2 und S3, die alle untereinander durch jeweils eine Kante verbunden sind. Ausgehend von S1 berechnet der Algorithmus von Dijkstra den kürzesten Pfad zu S2 und S3. Diese kürzesten Pfade sind als rote gestrichelte Linie in Abbildung 5.13 dargestellt und entsprechen den Kanten S1-S2 (Kosten: 3) und S1-S3 (Kosten: 5). Der Algorithmus von Kruskal hingegen berechnet den minimalen Spannbaum des Graphen, der aus den Kanten S1-S2 (Kosten: 3) und S2-S3 (Kosten: 3) besteht (dargestellt als blaue Linie in Abbildung 5.13). Der minimale Spannbaum hat somit ein Gewicht von $3 + 3 = 6$.

Die Algorithmen von Dijkstra und Kruskal werden von vielen weit verbreiteten Routing-Protokollen verwendet. So wird der Algorithmus von Dijkstra unter anderem von den Routing-Protokollen IS-IS (RFC 1142 [136]) und OSPF (RFC 2328 [137]) sowie einigen Routing-Protokollen für WSNs (siehe [138–140]) verwendet. Der Algorithmus von Kruskal hingegen ist besonders gut für Multicast-Routing geeignet, da er dafür sorgt, dass jeder Knoten die Nachricht erhält und hierbei als Route den minimalen Spannbaum des Netzwerkes verwen-

[4] Als Spannbaum wird ein Teilgraph eines Graphen bezeichnet, der ein Baum ist und alle Knoten des Graphen enthält [135]

[5] Ein minimaler Spannbaum besitzt das minimale Gewicht aller existierenden Spannbäume des Graphen [135]

det [141,142]. Bei der Umsetzung des ZRP-Protokolls wurde die Implementierung des Algorithmus von Dijkstra von Lars Vogel[6] sowie die Implementierung des Algorithmus von Kruskal von Prof. Dr. K. Schiele[7] zurückgegriffen.

5.7.5. Schnittstellen zum Routing-Stub

Die dritte Komponente des ZRP stellt die Schnittstelle zum Routing-Stub auf den Sensorknoten dar. Diese Schnittstelle ist dafür zuständig, auf eingehende Routenanfragen zu reagieren und die berechneten Routen an die anfragenden Sensorknoten zurückzuschicken. Auch diese Komponente betrifft lediglich diejenigen Knoten, die das IPv6-Protokoll implementieren. DPS-Client-Knoten, die den IPv6-Stub aus Abschnitt 5.6 verwenden, senden ihre ausgehenden IPv6-Pakete immer mittels der DPS-Verbindung zu ihrem Server.

Erhält ein Server ein IPv6-Paket von einem seiner Clients oder möchte er selbst ein Paket absenden, so überprüft er zunächst die Einträge in seiner Weiterleitungstabelle (Forwarding-Table). Befindet sich dort bereits ein Eintrag, wird die Nachricht an den entsprechenden benachbarten Sensorknoten gesendet, damit dieser die Nachricht zum Ziel weiterleitet. Befindet sich hingegen kein Eintrag in der Weiterleitungstabelle, sendet er eine Routenanfrage an das ZRP. Diese Routenanfrage beinhaltet seine eigene IPv6-Adresse sowie die IPv6-Adresse des Zielknoten.

Das ZRP konstruiert daraufhin aus den aktuellen Topologieinformationen einen Graphen des Netzwerkes und berechnet die kürzeste Route zwischen dem Sender und Empfänger. Die resultierende Route wird dann in die einzelnen Hops aufgeteilt und an die entsprechenden Routing-Stubs auf den Sensorknoten weitergeleitet. Möchte z.B. Knoten S1 eine Route nach S3 aufbauen und das ZRP ermittelt eine Route von S1 über S2 nach S3, so werden die folgenden Routen im Netzwerk eingetragen: „S1 kann S3 über S2 erreichen“ und „S2 kann S3 über S3 erreichen“. Auf diese Weise werden alle Sensorknoten entlang der Route mit den benötigten Informationen zur Weiterleitung der Nachricht versorgt. Die Sensorknoten empfangen die Route und speichern diese in ihrer lokalen Weiterleitungstabelle, wobei existierende Einträge für dieselbe Route überschrieben werden.

Da die ETX Metrik symmetrisch ist (d.h. sie gibt die Kosten der Verbindung in beiden Richtungen an) kann dieselbe Route auch für den Rückweg verwendet werden. Aus diesem Grund werden beide Richtungen der Route an die Sensorknoten weitergeleitet - zusätzlich zu „S3 kann S2 über S2 erreichen“ erhält S3 also auch den Eintrag „S3 kann S1 über S1 erreichen“.

[6] http://www.vogella.com/articles/JavaAlgorithmsDijkstra/article.html
[7] http://public.beuth-hochschule.de/~schiele/kruskal.html

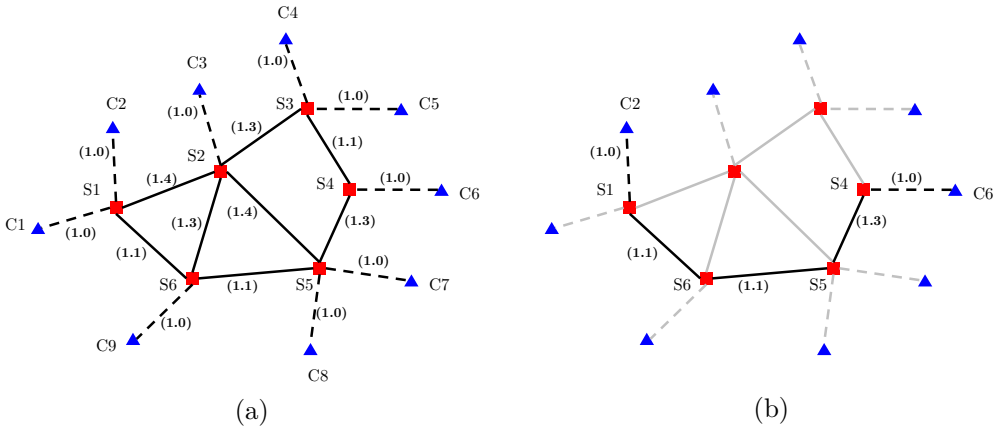


Abbildung 5.14.: Beispiel zur Berechnung einer Route zwischen den Clients C2 und C6 (Berechnete Route: C2-S1-S6-S5-S4-C6)

5.7.6. Beispiel

Der Ablauf und die Funktionsweise des ZRP wird in diesem Abschnitt anhand des Netzwerks in Abbildung 5.14 (a) erklärt, das aus sechs Server-Knoten (S1 bis S6, dargestellt als rote Rechtecke) und neun Client-Knoten (C1 bis C9, dargestellt als blaue Dreiecke) besteht. Die Client-Knoten haben sich bereits über das DPS-Protokoll mit jeweils einem Server-Knoten verbunden (dargestellt als gestrichelte Linien), z.B. sind C1 und C2 mit S1 verbunden. Die Links zwischen den Server-Knoten, die über Beacon-Meldungen des ZRP ermittelt wurden, sind als durchgezogene Linien dargestellt. Die Kosten aller Verbindungen im Netzwerk sind in Klammern an jeder Verbindung als gerundeter Wert in der ETX-Metrik angegeben (z.B. $ETX(S1, S2) = 1/(0{,}9 * 0{,}8) \approx 1{,}4$). Der ETX-Metrik der Verbindungen zwischen einem Client und einem Server wurde hierbei automatisch der optimale Wert 1,0 zugewiesen.

Diese Topologieinformationen werden vom ZRP als Adjazenzliste gespeichert und zur Konstruktion des Graphen verwendet. Zur Veranschaulichung ist diese Adjazenzliste in Form einer Adjazenzmatrix in Tabelle 5.2 dargestellt. Diese Darstellung wurde gewählt, da auf diese Weise die Verbindungen im Netzwerk für den Leser einfacher zu erkennen sind. Tabelle 5.2 beinhaltet keine Verbindungen zwischen zwei Clients, da zwei Clients immer über mindestens einen Server miteinander kommunizieren können (grauer Bereich unten rechts). Darüber hinaus ist zu erkennen, dass jeder Client genau eine Verbindung zu einem Server aufweist, während Server eine Verbindung zu einem oder mehreren Clients und Servern besitzen.

Im vorliegenden Beispiel soll nun der Client C2 ein IPv6-Paket an den Client C6 versenden. Hierfür verwendet C2 die DPS-Verbindung zu seinem Server S1, der das Paket empfängt und seine Weiterleitungstabelle überprüft. Da die Tabelle noch keinen Eintrag für eine Route nach C6 beinhaltet, sendet S1 eine Routenanfrage in der Form „Route von S1 nach C6“ an das ZRP. Das ZRP bestimmt nun zunächst den Server, der für C6 zuständig ist (S4) und berechnet

Tabelle 5.2.: Adjazenzmatrix des in Abbildung 5.14 dargestellten Netzwerkes

	S1	S2	S3	S4	S5	S6	C1	C2	C3	C4	C5	C6	C7	C8	C9
S1		1,4				1,1	1,0	1,0							
S2	1,4		1,3		1,4	1,3			1,0						
S3		1,3		1,1						1,0	1,0				
S4			1,1		1,3							1,0			
S5		1,4		1,3		1,1							1,0	1,0	
S6	1,1	1,3			1,1										1,0
C1	1,0														
C2	1,0														
C3		1,0													
C4			1,0												
C5			1,0												
C6				1,0											
C7					1,0										
C8					1,0										
C9						1,0									

anschließend den kürzesten Pfad zwischen S1 und S4. Dieser lautet: S1-S6-S5-S4 und ist in Abbildung 5.14 (b) dargestellt. Das ZRP teilt daraufhin allen Knoten entlang der Route den jeweils nächsten Knoten entlang der Route mit. Folglich erhält S1 die Antwort „S1 kann C6 über S6 erreichen“, während S6 die Information „S6 kann C6 über S5 erreichen“ erhält. S1 sendet das IP-Paket an S6, von wo das Paket über S5 an S4 weitergeleitet wird. Sobald das Paket bei S4 angekommen ist, erkennt dieser Server, dass es sich bei C6 um einen Client handelt, der eine DPS-Verbindung zu diesem Server aufgebaut hat. S6 sendet daraufhin das IP-Paket über die DPS-Verbindung an C6.

5.8. Wiselib-Implementierung

Neben der Implementierung des DPS-Protokolls für das iSense-Betriebssystem, die vom Autor dieser Arbeit durchgeführt wurde, wurde zusätzlich eine Implementierung des DPS-Protokolls für die Wiselib erstellt. Diese erfolgte im Rahmen des Google Summer of Code 2013 und wurde von einem Studenten unter der Betreuung durch den Autor dieser Arbeit durchgeführt. Im Folgenden werden zunächst der Google Summer of Code sowie die Wiselib kurz vorgestellt. Anschließend wird in Abschnitt 5.8.3 auf die Implementierung des DPS-Protokolls für die Wiselib eingegangen.

5.8.1. Google Summer of Code

Der Google Summer of Code (GSoC) ist ein Programmierstipendium, das seit 2005 jährlich von der Firma Google angeboten wird. Im Rahmen des GSoC erhalten Studenten die Möglichkeit, an einem Programmierprojekt teilzunehmen, das von einer Open-Source-Organisation betreut wird. Für die Bearbeitung des Projektes erhalten die Studenten eine finanzielle Unterstützung durch Google. Der Ablauf des GSoC ist hierbei wie folgt: Vor Beginn des GSoC können sich Open-Source-Organisationen in der sogenannten „Organization Application

Phase“ um eine Teilnahme am GSoC bewerben, indem sie bei Google eine Beschreibung ihrer Organisation und einer Menge an Programmierprojekten einreichen, die im Rahmen des GSoC bearbeitet werden sollen. Zusätzlich stellt jede Organisation eine Anzahl von Mentoren, welche die Arbeiten während des GSoC betreuen. Basierend auf diesen Bewerbungen entscheidet Google, welche Organisationen am GSoC teilnehmen dürfen.

Wurde eine Organisation für die Teilnahme ausgewählt, so werden die Programmierprojekte auf der Internetseite des GSoC veröffentlicht und die „Student Application Phase“ beginnt. Während dieser Phase wirbt die Organisation für ihre Projekte und Studenten können sich für die Bearbeitung der Projekte bewerben. Am Ende dieser Phase werden alle bei einer Organisation eingegangenen studentischen Bewerbungen von den Mentoren der Organisation bewertet. Basierend auf diesen Bewertungen und der Anzahl an Organisationen sowie Bewerbungen im gesamten GSoC weist Google daraufhin jeder Organisation eine Anzahl von Studenten zu, die diese Organisation annehmen darf. Die Organisation akzeptiert daraufhin maximal so viele Studenten, wie Google es ihr erlaubt hat und weist jedem Studenten einen oder mehrere Mentoren zu.

Im Anschluss beginnt die „Student Coding Phase“, der eigentliche Summer of Code. Während dieser Programmierphase bearbeiten die Studenten ihr Projekt, wobei sie von ihren Mentoren betreut werden. Dieser Vorgang ist vergleichbar zu einer Bachelorarbeit an einer Universität, wobei jedoch der Schwerpunkt bei der Implementierung und Dokumentation liegt. Im Gegensatz zu einer Bachelorarbeit müssen also keine Evaluation, Literaturrecherche oder eine schriftliche Ausarbeitung durchgeführt werden.

In der Mitte und am Ende der Programmierphase werden die Studenten von Ihrem jeweiligen Mentor bewertet. Falls der Student beide Bewertungen erfolgreich übersteht, erhält er ein Stipendium in Höhe von $5000 von Google. Die Organisation erhält eine Aufwandsentschädigung von $500 von Google für jeden Studenten, der am GSoC teilnimmt - unabhängig von den Bewertungen.

5.8.2. Wiselib

Die Wiselib ist eine quelloffene Algorithmen-Bibliothek für drahtlose Sensornetze [143], deren Entwicklung im Rahmen des WISEBED-Projekts begonnen wurde (siehe Abschnitt 2.1.6). Seit dem Abschluss des WISEBED-Projekts wird die Wiselib noch aktiv weiterentwickelt - unter anderem im Rahmen von mehreren studentischen Arbeiten im Google Summer of Code 2012 und 2013.

Die Wiselib beinhaltet eine Vielzahl von Algorithmen für Sensornetze, unter anderem Algorithmen zur Lokalisierung, Routing, Clustering, Graphenfärbung und Synchronisation. Darüber existiert seit dem GSoC 2012 eine 6LoWPAN- und IPv6-Implementierung, die in einer studentischen Arbeit um das Routing-Protokoll RPL erweitert wurde [144]. Die in der Wiselib implementierten Algorithmen kommunizieren über eine Abstraktionsschicht mit der jeweiligen Plattform, wodurch die Wiselib auf einer Vielzahl von unterschiedlichen Platt-

formen lauffähig ist. Dies beinhaltet vor allem die Sensornetz-Betriebssysteme Contiki, TinyOS und iSense, Netzwerksimulatoren wie NS3 und Shawn, aber auch die Plattformen Arduino, Android und Apple iOS.

Die Wiselib steht unter der GNU Lesser General Public License (LGPL Version 3 [145]), wodurch sie sowohl in andere Open-Source-Projekte als auch Closed-Source-Projekte sowie proprietäre Software eingebunden werden kann.

5.8.3. Durchführung

Die Implementierung des DPS-Protokoll für die Wiselib wurde im Rahmen des GSoC 2013 von Dániel Géhberger, einem Master Studenten von der Budapest University of Technology and Economics, durchgeführt. Das Projekt wurde vom Autor dieser Arbeit als Mentor betreut und in der Zeit vom Juni bis September 2013 bearbeitet. Dániel Géhberger hat bereits am GSoC 2012 für die Wiselib teilgenommen und unter der Betreuung des Autors dieser Arbeit den IPv6/6LoWPAN-Stack für die Wiselib implementiert und im Anschluss an den GSoC 2012 über dasselbe Thema seine Bachelorarbeit geschrieben [146].

Bereits bei der Implementierung des IPv6-Stack für die Wiselib traten dieselben Beschränkungen auf, die bereits in Abschnitt 5.5.2 für die iSense-Plattform beschrieben wurden: Die Quellcodegröße der IPv6-Implementierung inklusive dem Betriebssystem ist zu groß, um auf den stärker ressourcenbeschränkten Hardwareplattformen, wie z.B. dem JN5139 und dem TelosB ausgeführt zu werden. Plattformen mit mehr Programmspeicher, wie z.B. der JN5148, hingegen sind für den Einsatz der IPv6/6LoWPAN-Implementierung der Wiselib geeignet. Aus diesem Grund hat der Autor dieser Arbeit die Implementierung der in Abschnitt 5.7 beschriebenen Anwendung des DPS-Protokolls für die Wiselib für den GSoC 2013 vorgeschlagen und während des GSoC betreut. Hierbei wird der gesamte IPv6-Stack auf einem Knoten vom Typ JN5148 implementiert, der die IPv6-Schicht in Form eines Server-Skeletons über das DPS-Protokoll anbietet. Die stärker ressourcenbeschränkten Plattformen hingegen implementieren lediglich das DPS-Protokoll, den zugehörigen IPv6-Stub sowie das UDP- und ICMP-Protokoll. Weitere Details zu diesem Vorgehen finden sich in Abschnitt 5.7.

Durch die Open-Source-Implementierung für die Wiselib kann das DPS-Protokoll auf einer Vielzahl von unterschiedlichen Plattformen genutzt werden. Hieraus ergeben sich vielfältigere Einsatzmöglichkeiten des Protokolls, eine breitere potentielle Nutzerbasis sowie die Möglichkeit zur weiteren Evaluation des Protokolls. Als Referenz für die Implementierung wurden die über das DPS-Protokoll veröffentlichten Paper auf der CPScom 2012 [104] und CPScom 2013 [105] sowie den FGSN 2012 [147] verwendet. Zusätzlich wurden Auszüge aus dieser Arbeit verwendet, die zu einem Teil parallel zum GSoC entstanden sind. Neben der Verfügbarkeit des Protokolls für eine Vielzahl an zusätzlichen Hardwareplattformen zeigt die Implementierung für die Wiselib, dass die Spezifikation und Dokumentation des Protokolls den hierfür benötigten Umfang und die notwendige Qualität erreicht hat. Die im Rahmen der Evaluation in Abschnitt 6.5.3

und Abschnitt 6.6.2 vorgestellten Ergebnisse zeigen darüber hinaus, dass die beiden Implementierungen interoperabel sind und einen Vergleich erlauben.

Neben den oben genannten Vorteilen verfolgte die Wiselib-Implementierung das Ziel, die Leistung des DPS-Protokolls durch eine optimierte Implementierung zu erhöhen. Im Gegensatz zur Implementierung für die iSense-Plattform, die parallel zur Konzeption des DPS-Protokolls entstand und die kontinuierlich um optionale Funktionskomponenten erweitert wurde (z.B. die Header- oder die Payload-Kompression mittels LZW), sollte die Wiselib-Implementierung ausschließlich die aktuelle und in dieser Arbeit enthaltene Spezifikation umsetzen. Bereits bei der Umsetzung der IPv6/6LoWPAN-Implementierung für die Wiselib im GSoC 2012 konnte gezeigt werden, dass durch die Optimierung der Implementierung einerseits und den Einsatz der Wiselib andererseits eine Steigerung der Leistung des IPv6-Stack sowohl bei der Latenz als auch beim Datendurchsatz möglich ist. Eine genauere Darstellung dieses Sachverhaltes findet sich in der Evaluation in Abschnitt 6.5.3 und Abschnitt 6.6.2.

6. Evaluation

Nachdem im den vorangegangenen Kapitel die Implementierung des DPS-Protokolls vorgestellt wurde, beschäftigt sich dieses Kapitel mit der Evaluation des Protokolls. Zu Beginn des Kapitels wird in Abschnitt 6.1 untersucht, wie stark der Einsatz des DPS-Protokolls von der Art der verwendeten Knotenplatzierung abhängt. Hierbei wird zunächst auf die kontrollierte Knotenplatzierung eingegangen, zu der Untersuchungen anderer Forscher existieren, die sich auf das DPS-Protokoll übertragen lassen. Danach wird das DPS-Protokoll durch Simulationen in Szenarien mit zufälliger Knotenplatzierung untersucht, um festzustellen, wie sich die Aufteilung der Sensorknoten in Server und Clients auf die Konnektivität im Sensornetz auswirken.

Anschließend wird die Verwendung von IPv6 über das DPS-Protokoll experimentell mit der nativen 6LoWPAN-Implementierung der iSense-Plattform verglichen. Zusätzlich werden die Ergebnisse der Evaluationen von 6LoWPAN-Implementierungen für TinyOS herangezogen, die durch andere Forscher durchgeführt worden sind, um weitere Vergleichsmöglichkeiten neben der iSense-Plattform zu bieten. Außerdem wird die Implementierung des DPS-Protokolls für die iSense-Plattform mit der Implementierung für die Wiselib verglichen, die im Rahmen des GSoC 2013 entstanden ist (siehe Abschnitt 5.8).

Die Untersuchungen dieses Kapitels konzentriert sich auf einen quantitativen Vergleich der beiden Protokolle in einem Single-Hop-Szenario unter Laborbedingungen. Dies soll einen direkten Vergleich der beiden Protokolle ohne äußere Einflüsse ermöglichen. Ein Teil dieser Ergebnisse wurde im Rahmen eines Vortrags auf der CPScom 2012 [104] in Besançon vorgestellt. Einen Vergleich der beiden Protokolle in einem Anwendungsszenario unter realistischeren Rahmenbedingungen findet sich im anschließenden Kapitel 7.

Im Rahmen der Evaluationen in diesem Kapitel wird zunächst die Programmgröße untersucht, wobei insbesondere die Ersparnis auf Clientseite aber auch der Overhead auf Serverseite analysiert werden (siehe Abschnitt 6.3). Anschließend wird in Abschnitt 6.4 der Speicherverbrauch beim Senden und Empfangen von Nachrichten verglichen. Die Auswirkungen des DPS-Protokolls auf die Latenz werden durch eine Evaluation der ICMP-Round-Trip-Time in Abschnitt 6.5 untersucht, gefolgt vom maximalen Datendurchsatz beim Versenden von UDP-Paketen in Abschnitt 6.6 sowie der Paketankunftsrate in Abschnitt 6.7.

Eine Auswahl der in Abschnitt 4.3.4 vorgestellten Erweiterungen des DPS-Protokolls wird in Abschnitt 6.8 untersucht. Hierbei wird zunächst auf die Kompression des DPS-Headers eingegangen. Anschließend werden die Auswirkungen der Payload-Kompression auf die ICMP-Round-Trip-Time untersucht,

wobei der in Abschnitt 5.4 beschriebene S-LZW-Algorithmus zur Kompression verwendet wird. Den Abschluss der Evaluation bildet eine Zusammenfassung in Abschnitt 6.9.

6.1. Simulation: Knotenplatzierung

Die Platzierung der Sensorknoten innerhalb eines Netzes hat einen starken Einfluss auf die Eigenschafen und die Leistung des Sensornetzes. Im Rahmen des DPS-Protokolls ist z.B. darauf zu achten, dass ein Client nur dann mit dem Rest des Netzes kommunizieren kann, wenn er mit einem Server verbunden ist, der die Implementierung der hierfür notwendigen Kommunikationsprotokolle bereitstellt. Dieser Umstand muss bei der Planung und Ausbringung eines Sensornetzes berücksichtigt werden, welches das DPS-Protokoll einsetzen soll.

Die Knotenplatzierung kann laut [55] im Allgemeinen die folgenden Ziele verfolgen: das Sensornetz soll zusammenhängend sein, d.h. jeder Sensorknoten kann mit jedem anderen Sensorknoten (oder einer Untermenge, wie z.B. den Senken) kommunizieren (Network-Connectivity). Das Sensornetz soll darüber hinaus einen vorgegebenen Bereich mit seinen Sensoren erfassen können (Area-Coverage). In Abhängigkeit von den Anforderungen der Anwendung kann hierbei eine Überabdeckung gewünscht sein, z.B. um die Messergebnisse mehrerer Sensoren vergleichen zu können (Data-Fidelity), die Zuverlässigkeit der Kommunikation zu verbessern oder die Lebensdauer des Netzes zu erhöhen (Network-Longevity). Gleichzeitig soll die Anzahl der Sensorknoten im Netzwerk möglichst gering sein, um Kosten zu sparen.

Die Autoren von [55] beschreiben außerdem, dass die Ausbringung eines Sensornetzes auf zwei Arten erfolgen kann: Die kontrollierte Knotenplatzierung, bei der die Position aller Knoten im Netzwerk (mit einer bestimmten Genauigkeit) kontrolliert werden kann, sowie die zufällige Ausbringung, bei der keine Kontrolle über die Platzierung der einzelnen Knoten ausgeübt werden kann. Darüber hinaus existieren Mischformen, bei denen zunächst eine zufällige Ausbringung eines Teils der Sensorknoten erfolgt und anschließend durch die kontrollierte Positionierung der restlichen Knoten die Topologie an das gewünschte Ziel angepasst wird (vgl. RNP unten).

Die kontrollierte Ausbringung ist typisch für Sensornetze in Gebäuden, bei denen die Position der Sensoren in Abhängigkeit von der gewünschten Messgenauigkeit (z.B. ein Temperatursensor in jedem Raum, lückenlose Überwachung mittels Bewegungsmeldern) oder Netzabdeckung platziert werden. Bei der kontrollierten Ausbringung müssen jedoch auch Einschränkungen durch Hindernisse beachtet werden (Wände, Möbel). Auch Sensornetze außerhalb von Gebäuden können die kontrollierte Knotenplatzierung verwenden, wie z.B. das im Rahmen des Projekts FleGSens entwickelte Sensornetz zur Überwachung einer grünen Grenze.

Die zufällige Ausbringung hingegen ist typisch für Sensornetze in der freien Natur, bei denen die Knoten kurzfristig ausgebracht werden müssen, ohne dass

vorher eine genaue Planung durchgeführt werden konnte. Dies erfolgt z.B. beim Abwurf aus einem Flugzeug oder Hubschrauber oder bei der Ausbringung durch wenig geschultes Personal. Das Sensornetz dient in diesen Fällen häufig der Bekämpfung einer (Natur-)Katastrophe (z.B. Waldbrand oder Hochwasser) oder zur Überwachung eines Einsatzgebiets bei militärischen Anwendungen.

6.1.1. Kontrollierte Knotenplatzierung

Die Kontrollierte Knotenplatzierung wurde in der Literatur vielfach untersucht, um z.B. eine untere Schranken für die Anzahl an Sensorknoten zu bestimmen, mit denen ein gegebenes Gebiet beobachtet werden kann.

Eines der Probleme in diesem Zusammenhang stellt das Relay-Node-Placement (RNP) dar: Gegeben sei ein Sensornetz, das aus Sensorknoten und einer oder mehreren Basisstationen besteht. Die Sensorknoten sammeln Daten über ihre Umgebung, die zur Basisstation weitergeleitet werden sollen. Zusätzlich zu den Sensorknoten können nun Relay-Knoten dem Sensornetz hinzugefügt werden, um den Energieverbrauch der einzelnen Knoten zu reduzieren und die Lebensdauer des Netzes zu erhöhen. Es existieren hierbei zwei Unterkategorien des RNP-Problems: Das Single-Tiered-RNP (1T-RNP), bei dem sowohl die Sensorknoten als auch die Relay-Knoten-Nachrichten zur Basisstation weiterleiten können und das Two-Tiered-RNP (2T-RNP), bei dem nur die Relay-Knoten-Nachrichten weiterleiten können. Die Berechnung der optimalen Relay-Knoten-Platzierung ist komplex und sehr zeitaufwändig, und bereits 1999 konnte die NP-Schwere von 1T-RNP bewiesen werden [56], was ebenfalls für 2T-RNP vermutet wird [57].

Da die Sensorreichweite eines Knotens häufig niedriger ist als die Kommunikationsreichweite, spielt das 2T-RPN in diesen Fällen eine wichtige Rolle. In diesem Fall werden zunächst die Sensorknoten so ausgebracht, dass sie die zu beobachtende Fläche vollständig abdecken (Area-Coverage, s.o.), um anschließend durch das Hinzufügen von Relay-Knoten die Konnektivität des Netzes sicherzustellen. Da das 2T-RNP dem Knotenplatzierungsproblem des DPS-Protokolls entspricht (Sensorknoten entsprechen den Clients, Relay-Knoten entsprechen den Servern), wird im Folgenden auf das 2T-RNP eingegangen.

Das 2T-RNP kann in weitere Unterklassen unterteilt werden, bei denen das resultierende Netzwerk k-connected sein soll, um die Zuverlässigkeit des Netzes zu erhöhen, d.h. ein Sensorknoten ist immer in Reichweite von mindestens k Relay-Knoten (siehe [148–150]). Zu diesem Problem existieren mehrere α-Approximationsalgorithmen[1], die das Problem für $k \leqslant 2$ lösen, wobei unterschiedliche Bedingungen an die Funkreichweite der Relay-Knoten (F_R) und Sensorknoten (F_S) gestellt werden. So existiert eine 4,5-Approximation für k=1 und k=2 mit $F_R > 4 * F_S$ in [57] sowie eine $6 + \varepsilon$-Approximation für k=1 und $F_R = F_S$ in [149]. Da die Sensorknoten, die in den Experimenten in diesem und

[1] Ein α-Approximationsalgorithmus berechnet für ein gegebenes Minimierungs-Problem eine Lösung in Polynomialzeit, die im schlechtesten Fall um den Faktor α von der optimalen Lösung abweicht [151].

dem nachfolgenden Kapitel verwendet werden, vergleichbare Funkreichweiten besitzen ($F_R \approx F_S$), kann der zuletzt genannte α-Approximationsalgorithmus auf das Knotenplatzierungsproblem im Rahmen des DPS-Protokolls angewendet werden.

6.1.2. Zufällige Knotenplatzierung

Wie im vorangegangenen Abschnitt beschrieben wurde, existieren in der Literatur bereits viele Untersuchungen über das 2T-RNP Problem, die Algorithmen zu dessen effizienter Approximation bereitstellen. Da das DPS-Protokoll für Szenarien mit kontrollierter Knotenplatzierung konzipiert wurde, können diese Algorithmen folglich zur automatischen Planung dieser Szenarien verwendet werden. In diesem Abschnitt soll hingegen untersucht werden, welchen Einfluss eine zufällige Platzierung der Knoten auf die Konnektivität der erzeugten Topologie hat. Als Konnektivität wird in diesem Zusammenhang die Verbindung eines Sensorknotens mit der Basisstation bezeichnet.

Szenarien mit zufälliger Knotenplatzierung können im Rahmen von Simulationen untersucht werden, bei denen ein gegebenes Protokoll in einer Vielzahl von zufällig generierten Szenarien untersucht wird. In der Arbeit von Ishizuka und Aida [152] werden hierfür insgesamt drei verschiedene Verteilungen (Simple Diffusion, Uniform und R-Random) vorgestellt, bei denen die Knoten des Netzwerks jeweils zufällig um die Basisstation herum angeordnet werden. Die Autoren von [152] urteilen in ihrer Zusammenfassung, dass die R-Random Verteilung die größte Robustheit bezüglich Knotenausfall besitzt und deshalb in der Realität bevorzugt eingesetzt werden sollte, weshalb diese für die Simulationen in diesem Abschnitt verwendet wird.

Die simulative Untersuchung des DPS-Protokolls in diesem Abschnitt soll dazu dienen, die Konnektivität der Sensorknoten im Netzwerk in Abhängigkeit von dem Verhältnis von Servern zu Clients zu analysieren. Die Abdeckung der Simulationsfläche durch die Sensorik der Sensorknoten wird hierbei nicht untersucht. Als Simulationsszenario wird eine Anwendung gewählt, bei der alle Sensorknoten im Netzwerk ihre Daten an eine Basisstation schicken, die sich im Zentrum des Netzwerks befindet. Für die Untersuchung der Konnektivität ist also von Interesse, ob jeder Server (über einen oder mehrere Hops) mit der Basisstation kommunizieren kann und ob jeder Client mindestens einen Server in Funkreichweite besitzt, der diese Eigenschaft besitzt. Die Untersuchungen in diesem Abschnitt verwenden das Unit-Disk-Graph-Modell und gehen von symmetrischen Links aus, um eine potentielle Verbindung zwischen zwei benachbarten Sensorknoten zu erkennen. Dieses Modell trifft keine Aussage über die Qualität der Verbindung oder die Empfangswahrscheinlichkeit. Diese Simulationen sollen untersuchen, ob eine Verbindung zwischen benachbarten Sensorknoten möglich ist.

Für die Simulationen wurde ein Java-Programm verwendet, das die folgenden Parameter verwendet: Die simulierte Fläche besitzt eine Größe von 100 x 100 m,

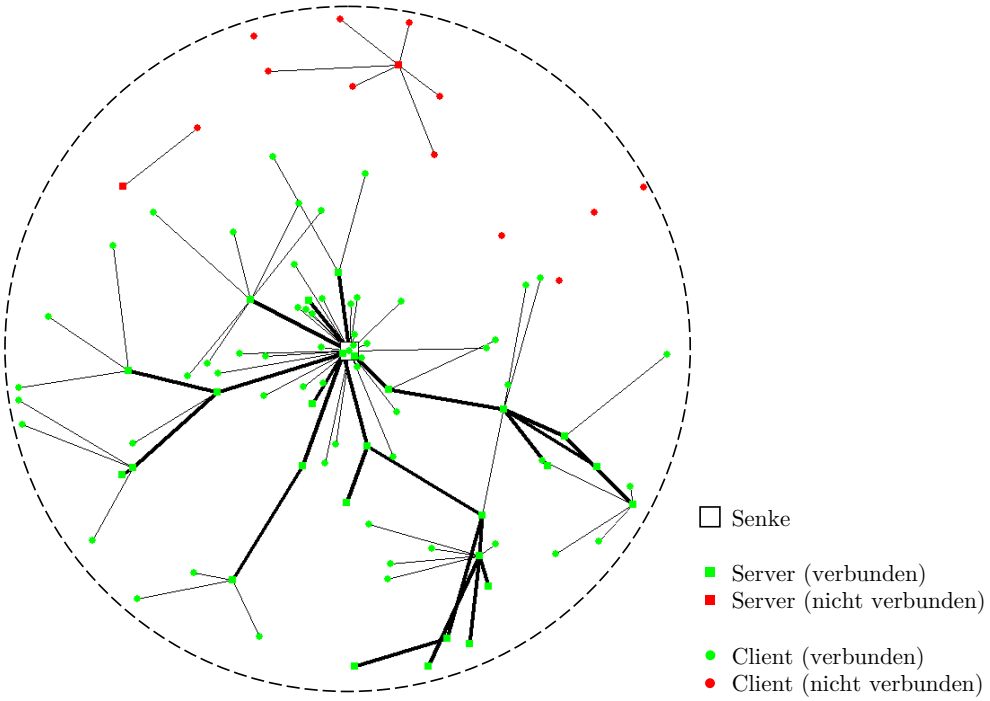


Abbildung 6.1.: Beispiel für eine simulierte Netzwerktopologie (100 Sensorknoten, 30 % Server, 100 x 100 m, Funkreichweite 20 m)

in dessen Zentrum sich die Basisstation befindet. Die Kommunikationsreichweite aller Knoten im Netz beträgt 20 m ($F_R = F_S$). Auf der Fläche werden zusätzlich zur Basisstation N-1 Sensorknoten mittels der R-Random Verteilung verteilt. Ein Beispiel für eine der erzeugten Topologie findet sich in Abbildung 6.1. Anschließend wird eine zufällige Untermenge der Sensorknoten zu Servern erklärt (dargestellt als Rechtecke in Abbildung 6.1), während die verbliebenen Knoten zu Clients erklärt werden (dargestellt als Kreise in Abbildung 6.1). In dem resultierenden Sensornetz wird nun ausgehend von der Basisstation für alle Server eine Routing-Topologie mittels des Dijkstra-Algorithmus erstellt (dicke schwarze Kanten). Diese Topologie verbindet alle Server mit der Basisstation, falls möglich.

Anschließend wird für alle Clients im Netz ermittelt, ob sich ein Server in Funkreichweite befindet. Falls sich mehrere Server in Funkreichweite eines Knotens befinden, wird derjenige ausgewählt, der mit der Basisstation verbunden ist und dessen Entfernung zum Client am niedrigsten ist. Anschließend wird eine Färbung der Knoten vorgenommen, wobei alle Knoten, die mit der Basisstation verbunden sind, die Farbe Grün erhalten und alle Knoten, die nicht mit der Basisstation verbunden sind, die Farbe Rot erhalten. In der Beispieltopologie in Abbildung 6.1 sind 76 der 100 Knoten mit der Basisstation verbunden, wobei zwei Server (oben Mitte und oben links) nicht mit der Basisstation über andere Server verbunden sind und insgesamt zwölf Clients entweder keinen Server in Funkreichweite besitzen oder lediglich einen der beiden Server, die nicht mit der Basisstation verbunden sind.

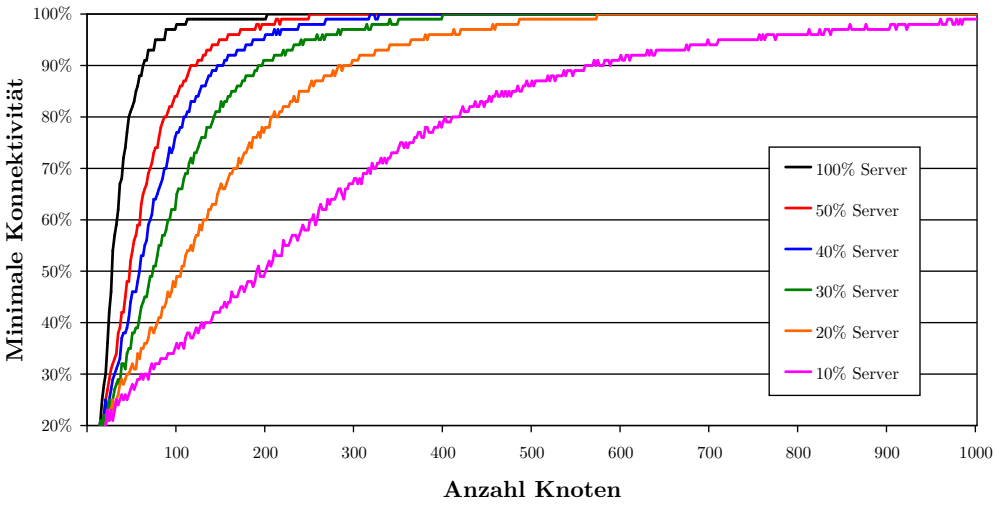


Abbildung 6.2.: Untersuchung der Konnektivität bei zufälliger Knotenplatzierung in Abhängigkeit vom Server-Anteil

Dieser Vorgang wurde im Rahmen der Simulationen mit unterschiedlichen Netzwerkgrößen von 2 bis 1.000 Knoten wiederholt, wobei zusätzlich der Anteil der Server variiert wurde (von 10 % bis 50 % in Schritten von 10 % sowie zusätzlich 100 %). Für jede dieser Konfigurationen wurden jeweils 10.000 Simulationen mit unterschiedlichen Knotenanordnungen (Seeds) durchgeführt. Für jede Simulation wurde die Konnektivität ermittelt, d.h. der Anteil der Knoten, die mit der Basisstation verbunden sind. Die Ergebnisse der Simulationen finden sich zusammengefasst in Abbildung 6.2, die für den gegebenen Server-Anteil die minimale Konnektivität angibt, die in 99 % aller Fälle erreicht wurde. Für das Beispiel aus Abbildung 6.1 mit 100 Sensorknoten und 30 % Servern bedeutet dies, dass in 99 % aller Simulationen eine Konnektivität von mindestens 65 % erreicht wurde, während nur in lediglich 1 % der Simulationen eine schlechtere Konnektivität erreicht wurde. Anders ausgedrückt befinden sich 99 % aller Simulationsergebnisse oberhalb der in Abbildung 6.1 dargestellten Kurven, während sich nur 1 % unterhalb befindet. Das Beispiel in Abbildung 6.1 mit einer Konnektivität von 76 % gehört folglich zu den oberen 99 %. Zum Vergleich: Die niedrigste beobachtete Konnektivität bei einem Server-Anteil von 30 % und einer Knotenanzahl von 100 beträgt 39 %. Die in Abbildung 6.2 dargestellten Ergebnisse zeigen, dass die Konnektivität mit steigender Knotenanzahl zunimmt und sich asymptotisch einem Wert von 100 % annähert. Hierbei nimmt die Geschwindigkeit der Annäherung ab, je niedriger der Server-Anteil ausfällt und je höher die Konnektivität ist: Bei 100 % Server-Anteil sind lediglich 112 Knoten nötig, um in 99 % aller Fälle eine Konnektivität von 99 % zu erreichen, während hierfür 212 Knoten bei 50 % Server-Anteil notwendig sind oder sogar 960 Knoten bei einem Server-Anteil von nur 10 %.

Eine genauere Analyse der durch die Simulationen gewonnenen Daten findet sich in dem Box-Whisker-Plot in Abbildung 6.3. In dieser Abbildung wurden die Ergebnisse von der Anzahl der Knoten im Netzwerk auf die resultierende Netzwerkdichte verallgemeinert. Die Dichte eines Netzwerks wird hierbei nach

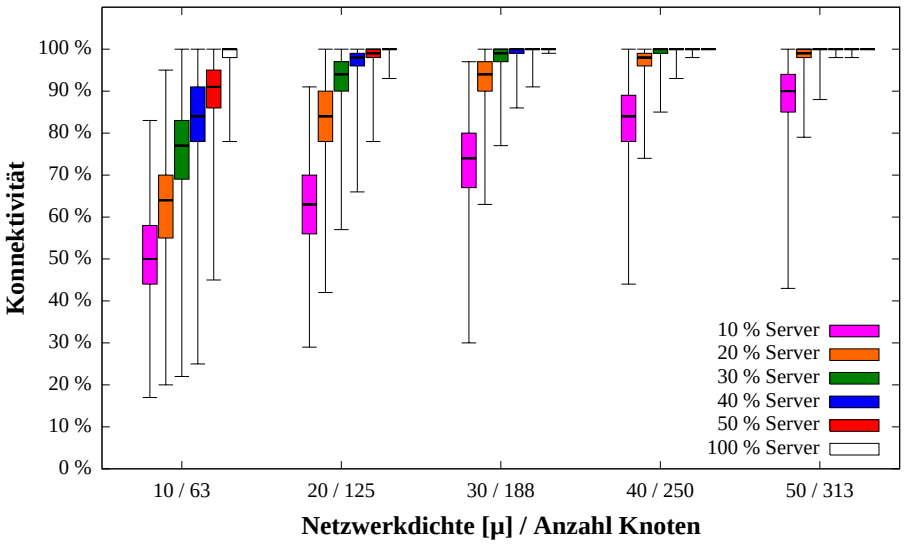


Abbildung 6.3.: Box-Whisker-Plot der Konnektivität bei zufälliger Knotenplatzierung in Abhängigkeit vom Server-Anteil (Whisker stellen Minimum und Maximum dar)

der folgenden Formel berechnet: $\mu = \frac{N\pi R^2}{A}$ [153], wobei N der Anzahl der Sensorknoten im Netzwerk entspricht, A dem Flächeninhalt des Netzes und R der Funkreichweite der Knoten. Die Netzwerkdichte gibt also an, wie viele Sensorknoten sich durchschnittlich im Kommunikationsradius jedes Sensorknoten befinden. In unseren Simulationen ergeben sich die folgenden Werte: N = 2 bis 1000, $A = \pi * (\frac{100\,m}{2})^2 = 7854\,\mathrm{m}^2$ und $R = 20\,\mathrm{m}$. Die Ergebnisse in Abbildung 6.3 stellen hierbei die Konnektivität für Netzwerkdichten zwischen μ=10 (63 Knoten) und μ=50 (313 Knoten) bei einem Server-Anteil zwischen 10 % und 50 % im Vergleich zum Optimum mit einem Server-Anteil von 100 % dar. Wie zu erwarten ist auch hierbei zu erkennen, dass die Konnektivität mit steigender Netzwerkdichte zunimmt, wobei diese für Szenarien mit höherem Server-Anteil schneller zunimmt, als mit niedrigerem Server-Anteil. Der Box-Whisker-Plot stellt sowohl den Median der Daten (schwarzer Balken in der Box), das obere und untere Quartil (das obere und untere Ende der Box) sowie das Minimum und Maximum der gesammelten Daten dar (unterer und oberer Whisker). Während sich innerhalb der Box 50 % aller Messwerte befinden, befinden sich oberhalb des unteren Quartils 75 % aller Messwerte.

Die in Abbildung 6.3 dargestellten Simulationsergebnisse geben einen Anhaltspunkt für Sensornetzbetreiber, die das DPS-Protokoll in Szenarien mit zufälliger Knotenplatzierung einsetzen wollen. So kann der Betreiber entscheiden, welcher Anteil der Sensorknoten als Server für das DPS-Protokoll eingesetzt werden soll, um anschließend über die Netzwerkdichte die gewünschte Konnektivität zu erreichen. Obwohl das DPS-Protokoll nicht für Szenarien mit zufälliger Knotenplatzierung konzipiert wurde, lässt sich auf diese Weise bei einem Server-Anteil von 30 % und einer Netzwerkdichte von μ =20 in 75 % aller Fälle eine Konnektivität von mindestens 90 % erreichen. Wird die Netzwerkdichte auf μ =30 erhöht,

kann in 75 % aller Fälle eine Konnektivität von mindestens 97 % und in 50 % aller Fälle sogar eine Konnektivität von mindestens 99 % erreicht werden. Zusätzlich ist aus den Ergebnissen zu entnehmen, dass das DPS-Protokoll bei zufälliger Knotenplatzierung und einem Server-Anteil von 10 % (oder weniger) nur eine vergleichsweise geringe Konnektivität erreicht, weshalb in diesen Fällen das DPS-Protokoll nicht geeignet erscheint. Dies ist insbesondere an der minimalen Konnektivität zu erkennen, die bei einer Netzwerkdichte von μ =50 lediglich 43 % beträgt.

Da die Sensorreichweite in vielen Fällen geringer als die Kommunikationsreichweite ist, muss in diesen Fällen ohnehin eine höhere Netzwerkdichte gewählt werden, um die Sensorabdeckung des Netzes zu gewährleisten. So wird in der Arbeit von [154] bewiesen, dass ein Sensornetz, bei dem die Kommunikationsreichweite mindestens dem doppelten der Sensorreichweite entspricht, die Sensorabdeckung (1-coverage) des Gebiets die Konnektivität aller Sensorknoten garantiert.

6.2. Experimente: Versuchsaufbau

In den nachfolgenden Abschnitten wird das DPS-Protokoll hinsichtlich mehrerer Leistungsmerkmale hin untersucht. Diese Experimente dienen zunächst der Evaluation des DPS-Protokolls in einem Single-Hop-Szenario unter Laborbedingungen. Der in diesem Abschnitt beschriebene Versuchsaufbau gilt für alle Experimente in diesem Kapitel. Die Multi-Hop-Evaluation verwendet einen anderen Versuchsaufbau, der im nächsten Kapitel in Abschnitt 7.2 beschrieben wird.

Als Hardwareplattform für die Evaluation werden iSense-Sensorknoten der Firma coalesenses verwendet. Diese sind entweder mit dem JN5139- oder dem JN5148-Mikrocontroller ausgestattet, die beide mit einer Geschwindigkeit von 16 MHz getaktet sind und eine 32-Bit RISC-Architektur verwenden. Während der JN5139 über lediglich 96 kB RAM-Speicher verfügt (wovon nur 92 kB zur Laufzeit verwendet werden können), ist sein Nachfolger, der JN5148, mit 128 kB RAM-Speicher ausgestattet. Dieser Speicher wird zur Laufzeit sowohl für die Speicherung des Programms als auch für den Heap und Stack verwendet.

Für die Single-Hop-Evaluation wurden drei Sensorknoten in einem Dreieck mit etwa 20 cm Kantenlänge in einer Höhe von etwa 1 m über dem Boden auf einem Tisch ausgelegt, was in Abbildung 6.4 dargestellt ist. Bei den Sensorknoten handelt es sich um zwei DPS-Server vom Typ JN5148 und um einen DPS-Client vom Typ JN5139. Auf den drei Sensorknoten läuft das iSense-Betriebssystem (Stand: 04/2012), das um die in Abschnitt 5.6 beschriebenen Funktionen erweitert wurde: Die Server teilen ihre IPv6-Schicht in Form des Server-Skeletons über das DPS-Protokoll mit dem Client, der lediglich den IPv6-Stub sowie das UDP- und ICMP-Protokoll implementiert. Die Funkkommunikation erfolgt auf Kanal 26 des 802.15.4-Protokolls, der sich nicht mit den vom WLAN der Universität verwendeten Kanälen überschneidet. Neben den für die Evaluation verwendeten Nachrichten werden keinerlei weiteren Nachrichten zwischen den Sensorknoten

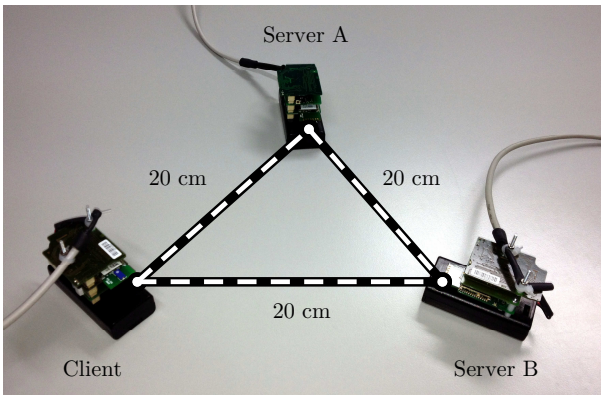


Abbildung 6.4.: Versuchsaufbau der Single-Hop-Experimente: Ein Client und zwei Server in einem Dreieck mit ca. 20 cm Kantenlänge

ausgetauscht. Es wird kein Routing-Protokoll verwendet, sondern stattdessen eine statische Routingtabelle, die nur Single-Hop-Kommunikation zwischen den drei Sensorknoten vorsieht.

6.3. Programmgröße

Die Motivation hinter dem DPS-Protokoll liegt vor allem in der Programmgröße vieler existierender IPv6-Implementierungen begründet, die den Einsatz von IPv6 auf stärker ressourcenbeschränkten Geräten einschränkt oder verhindert. Die Größe der einzelnen Komponenten des für die Single-Hop-Experimente verwendeten Programms sind in Tabelle 6.5 aufgelistet. Diese Daten wurden aus den kompilierten Objektdateien unter Verwendung des Programms *nm* erzeugt, das alle Symbole und deren Größe aus den Objektdateien auflisten kann. Die Daten wurden anschließend manuell den einzelnen Klassen zugeordnet und aufsummiert. Die in diesem Abschnitt vorgestellten Experimente verwenden die in Abschnitt 6.2 ausgewählten Plattformen als Beispiele, die Ergebnisse lassen sich jedoch auf andere Hardwareplattformen übertragen. Dies wird unter anderem an den Programmgrößen der Contiki-Implementierungen in Tabelle 6.5 deutlich.

Die Ergebnisse in Tabelle 6.5 zeigen hierbei, dass die Evaluations-Anwendung zusammen mit dem nativen IPv6-Stack nicht auf der ressourcenbeschränkteren JN5139-Plattform ausgeführt werden kann, da hierfür insgesamt 104,2 kB Programmspeicher benötigt werden, während auf dieser Plattform lediglich 92 kB Speicher zur Verfügung stehen (der zur Laufzeit zusätzlich noch für den Heap und den Stack verwendet werden muss). Auf der JN5148-Plattform stehen insgesamt 128 kB Speicher zur Verfügung, während dieselbe Anwendung lediglich 67,3 kB Programmspeicher belegt, wodurch die Evaluations-Anwendung auf dieser Plattform ausgeführt werden kann. Der Größenunterschied zwischen den beiden Plattformen ergibt sich durch eine verkleinerte Firmware und einen verbesserten Compiler: Der neuere ba-elf-ba2 Compiler der JN5148-Plattform

	iSIPS (JN5148)	DPS-Server (JN5148)	iSIPS (JN5139)	DPS-Client (JN5139)
Anwendung	3,8 kB		5,5 kB	
Betriebssystem	36,4 kB		48,5 kB	
IPv6-Stack	27,1 kB		50,2 kB	9,9 kB
$\sum$ iSense	67,3 kB		104,2 kB	63,9 kB
DPS-Protokoll	–	8,5 kB	–	15,5 kB
Bestätigungen	–	0,8 kB	–	1,5 kB
Heartbeats	–	0,6 kB	–	1,2 kB
AES-Prüfsummen	–	0,3 kB	–	0,3 kB
$\sum$ DPS	–	10,2 kB	–	18,5 kB
$\sum$ Insgesamt	67,3 kB	77,5 kB	104,2 kB	82,4 kB
Relativ	100 %	+15 %	100 %	-20 %
		+10,2 kB		-21,8 kB

Abbildung 6.5.: Vergleich der Programmgröße der Bestandteile der iSIPS-Implementierung und des DPS-Protokolls

unterstützt Opcodes mit variabler Länge, wodurch die Programmgröße im Vergleich zum älteren ba-elf Compiler der JN5139-Plattform optimiert werden konnte.

Durch den Einsatz des DPS-Protokolls konnte die Größe des IPv6-Stack für den Client von 50,2 kB auf 9,9 kB verringert werden, da der Client lediglich den IPv6-Stub, das UDP-Protokoll und das ICMP-Protokoll implementieren muss. Im Gegenzug belegt das DPS-Protokoll auf dieser Plattform 18,5 kB, so dass die Programmgröße insgesamt um 20 % auf 82,4 kB reduziert werden konnte, was den Betrieb der Anwendung zusammen mit dem IPv6-Stack ermöglicht. Auf der Seite des DPS-Servers vergrößert sich der benötigte Programmspeicher um 10,2 kB auf 77,5 kB, was der Größe des DPS-Protokolls inklusive dem IPv6-Skeleton auf dieser Plattform entspricht. Dieser geringfügige Anstieg der Anforderungen an den Programmspeicher von lediglich 15 % für den DPS-Server ermöglicht eine Reduktion von 20 % für den DPS-Client und erlaubt somit den Betrieb der Anwendung und des IPv6-Stacks auf beiden Plattformen in einem heterogenen Netzwerk.

Um die Ergebnisse auf andere Hard- und Softwareplattformen übertragen zu können, wurden die Programmgrößen der Contiki-IPv6-Implementierung aus Tabelle 3.1 (Seite 30) für das hier beschriebene Szenario übernommen. Die Gesamtgröße der IPv6-Implementierung von Contiki beträgt 29,3 kB auf dem MSP430 und 48,1 kB auf dem JN5139. Dies ist in Tabelle 6.6 dargestellt, die ebenfalls eine Abschätzung der Größe eines DPS-Clients auf diesen Plattformen beinhaltet. Da die Contiki-Portierung für den JN5139 den selben Compiler und die selbe Programmiersprache wie das iSense-Betriebssystem verwendet, wird an dieser Stelle die Annahme getroffen, dass eine Portierung des DPS-Protokolls für diese Plattform eine Programmgröße von ebenfalls 18,5 kB aufweisen wird. Die Programmgröße des DPS-Protokolls für die MSP430-Plattform wurde er-

	Contiki (MSP430)	Contiki DPS-Client (MSP430)	Contiki (JN5139)	Contiki DPS-Client (JN5139)
6LoWPAN	4,6 kB	–	8,0 kB	–
IPv6	7,4 kB	–	12,3 kB	–
ND	6,8 kB	–	13,3 kB	–
Routing	9,0 kB	–	12,9 kB	–
UDP	0,7 kB		0,3 kB	
ICMP	0,8 kB		1,3 kB	
$\sum$ IPv6-Stack	29,3 kB	1,5 kB	48,1 kB	1,6 kB
DPS-Protokoll	–	18,7 kB*	–	18,5 kB*
$\sum$ Insgesamt	29,3 kB	20,2 kB	48,1 kB	20,1 kB
Relativ	100 %	-36 %	100 %	-58 %
		-9,1 kB		-28,0 kB

* Siehe Anmerkungen im Text

Abbildung 6.6.: Vergleich der berechneten Programmgröße der Contiki-IPv6-Implementierungen mit dem DPS-Protokoll

mittelt, indem die iSense-Version des DPS-Protokolls mittels des MSP430-GCC-Compilers für den MSP430 compiliert wurde. Die Programmgrößen des DPS-Protokolls für die Contiki-Plattform stellen somit für beide Hardwareplattformen nur eine Schätzung dar. Da jedoch dieselbe Programmiersprache und dieselben Compiler verwendet werden, ist keine große Abweichung von den hier angegebenen Größen zu erwarten.

Aus den in Tabelle 6.6 dargestellten Daten kann gefolgert werden, dass die Reduktion der Programmgröße auch auf andere Plattformen und Betriebssysteme übertragen werden kann. So kann die Programmgröße des IPv6-Stacks für den DPS-Client um etwa 36 % (MSP430) bzw. um bis zu 58 % (JN5139) reduziert werden. Diese Werte sind vergleichbar zu der für die iSense-Plattform auf dem JN5159 erreichten Reduktion des IPv6-Stacks um 43 % (50,2 kB auf 28,4 kB, vgl. Tabelle 6.5).

6.4. Speicherverbrauch

Neben der Programmgröße einer Implementierung ist der Speicherverbrauch der Anwendung zur Laufzeit von Interesse. Im Gegensatz zur statischen Programmgröße gibt diese den dynamischen Speicherverbrauch zur Laufzeit an, der sich in Abhängigkeit von der Größe der zu sendenden oder empfangenen Nachrichten verändern kann. Aus diesem Grund wird in diesem Abschnitt zunächst der Verbrauch der einzelnen Programmbestandteile im Ruhezustand analysiert und anschließend der Speicherverbrauch beim Senden und Empfangen von Nachrichten unterschiedlicher Größe verglichen.

	DPS-Server (JN5148)	DPS-Client (JN5139)	
IPv6-Stack	2312 Byte	148 Byte	-2164 Byte (-93,6 %)
DPS Protokoll	772 Byte	772 Byte	
DPS Verbindung	92 Byte	92 Byte	
Betriebssystem	4280 Byte	4272 Byte	
$\sum$	7456 Byte	5284 Byte	-2172 Byte (-29,1 %)
Freier Speicher	37920 Byte	2604 Byte	

Abbildung 6.7.: Vergleich des Speicherverbrauchs zur Laufzeit zwischen DPS-Server und -Client

Um den Speicherverbrauch der verschiedenen Programmbestandteile im Ruhezustand zu ermitteln, verwendet die Evaluations-Anwendung die `isense_memory` Komponente des iSense-Betriebssystems, die den Heap zur Laufzeit verwaltet und die über die Methode `mem_used()` die Größe des aktuell verwendeten Speichers anzeigt. Die Evaluations-Anwendung gibt nun während des Startvorgangs sowie in periodischen Abständen zur Laufzeit den jeweils aktuell verwendeten Speicher aus. Zusätzlich wurde die `isense_memory` Komponente erweitert, so dass diese neben dem aktuellen Speicherverbrauch den maximal verwendeten Speicher zur Laufzeit aufzeichnet. Die Ergebnisse dieser Untersuchungen finden sich zusammengefasst in Tabelle 6.7 sowie in Abbildung 6.8.

In Tabelle 6.7 ist der Speicherverbrauch der einzelnen Programmbestandteile auf dem DPS-Server sowie dem DPS-Client aufgelistet: In beiden Fällen benötigt das Betriebssystem etwa 4,2 kB RAM. Dies beinhaltet sämtliche sonstigen Programmbestandteile wie die Sensoren oder Funkschnittstelle. Das DPS-Protokoll hingegen benötigt in beiden Fällen 772 Byte, die sich um zusätzliche 92 Byte für jede aktive DPS-Verbindung erhöhen. Während der Client zur Laufzeit höchstens eine DPS-Verbindung verwendet, könnte der Server in der beschriebenen Konfiguration theoretisch bis zu 37920 Byte / 92 Byte = 412 DPS-Verbindungen aufbauen. Auf dem DPS-Server benötigt der IPv6-Stack zur Laufzeit 2312 Byte, was durch den Einsatz des DPS-Protokolls auf 148 Byte auf der Seite des DPS-Clients reduziert werden kann. Hierdurch verringert sich der Speicherverbrauch des IPv6-Stack durch die Verwendung des IPv6-Stub um 93,6 %, wodurch der Gesamtspeicherverbrauch um 29,1 % reduziert werden kann.

Um den Speicherverbrauch beim Senden und Empfangen von Nachrichten zu vergleichen, wurden zwischen jeweils zwei Sensorknoten UDP-Pakete mit unterschiedlichen Payload-Längen versendet. Währenddessen wurde sowohl auf dem Sender als auch auf dem Empfänger in periodischen Abständen der maximale Speicherverbrauch aufgezeichnet. Die Ergebnisse sind in Abbildung 6.8 dargestellt: Aus den hier zusammengefassten Daten ist zu erkennen, dass sich der Speicherverbrauch beim Senden oder Empfangen einer Nachricht aus insgesamt drei Teilen zusammensetzt: Dem Speicherverbrauch im Ruhezustand (MEM_0), einem Grundverbrauch pro Nachricht (MEM_1) sowie einem dynamischen Anteil, der sich in Abhängigkeit von der Payload-Länge erhöht (MEM_P). Hierbei entspricht MEM_0 den in Tabelle 6.7 dargestellten Werten zuzüglich

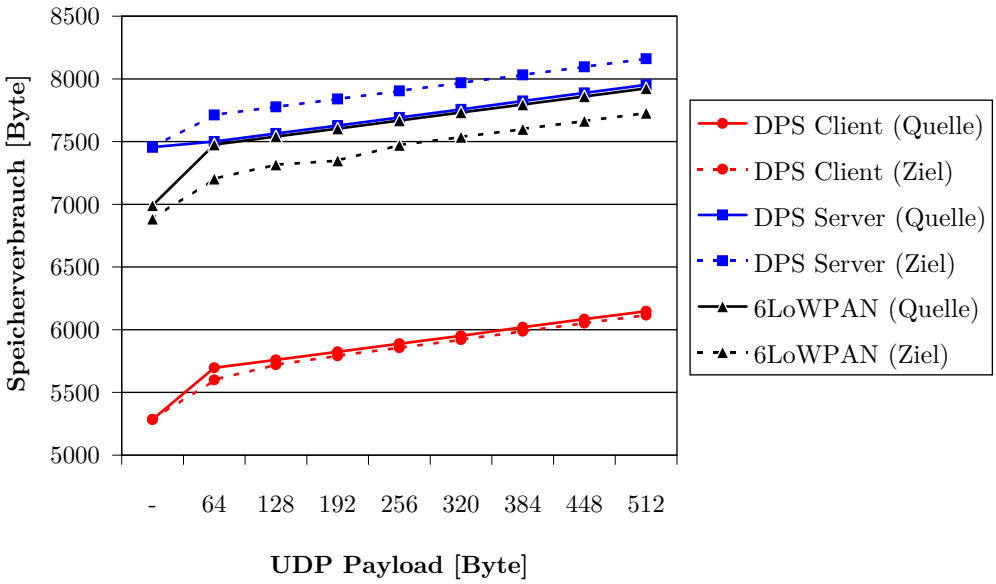


Abbildung 6.8.: Vergleich des Speicherverbrauchs beim Senden und Empfangen von UDP-Paketen in Abhängigkeit von der Payload-Länge

des Speicherverbrauchs der regelmäßigen Heartbeat-Nachrichten. MEM_1 hingegen entspricht dem Protokoll-Overhead beim Versenden eines UDP-Pakets, der durch die Header des DPS-, IP- und UDP-Protokolls sowie die entsprechenden Datenstrukturen entsteht. Anschließend steigt der Speicherverbrauch linear an, wobei MEM_P in allen Fällen genau einem Byte pro Payload-Byte entspricht. Abseits der unterschiedlich hohen Werte für MEM_0 sowie MEM_1 bedeutet dies, dass durch das DPS-Protokoll kein weiterer Overhead beim Versenden oder Empfangen von Nachrichten entsteht.

Wie bereits bei der Programmgröße profitiert der Client auch beim Speicherverbrauch durch den Einsatz des DPS-Protokolls zu Lasten des Servers. In diesem Fall erhöht sich der Speicherverbrauch des Servers beim Nachrichtenempfang um 432 Byte im Vergleich zum nativen 6LoWPAN-Fall, wohingegen der Speicherverbrauch beim Versenden einer Nachricht nur um 28 Byte steigt. Bei einer Gesamtmenge von über 37 kB freiem Speicher auf dem Server hat dieser Overhead keinen nennenswerten Einfluss auf dessen Leistungsfähigkeit. Der Einsatz des DPS-Protokolls ermöglicht es dem Client im Gegenzug, UDP-Pakete mit dem Server und dem restlichen Netzwerk auszutauschen. Aus den in Abbildung 6.8 dargestellten Daten ergibt sich als Hochrechnung, dass es für den Client möglich ist, UDP-Pakete mit einer maximalen Payload-Länge von 2200 Byte zu senden und zu empfangen. Dies konnte experimentell bis zu einer maximalen Payload-Länge von 2000 Byte nachgewiesen werden, da bei dieser Payload-Länge das resultierende IPv6-Paket inkl. des IP- und UDP-Headers die maximale DPS-Nachrichtenlänge von 2048 Byte erreicht (siehe Abschnitt 4.2). Dieser Wert ist größer als die für IPv6 in RFC 2460 [8] geforderte minimale MTU von 1280 Byte und der empfohlenen MTU von 1500 Byte. Darüber hinaus ist dieser Wert größer als die von anderen IPv6-Implementierungen für

Sensorknotenplattformen standardmäßig verwendeten Größen: uIPv6 unter Contiki verwendet standardmäßig Werte zwischen 140 und 450 Byte und maximal 1300 Byte.

6.5. Latenz

Eine weitere wichtige Eigenschaft eines Netzwerkprotokolls ist die Latenz, die häufig in Form der Round-Trip-Time (RTT) gemessen wird. Hierbei sendet ein Sender eine Nachricht an einen Empfänger und misst die Zeit bis zum Empfang einer Antwortnachricht vom Empfänger. Die RTT hängt hierbei von vier Faktoren ab:

- Die **Übertragungsverzögerung** wird durch die zur Verfügung stehende Bandbreite und die Länge der zu sendenden Nachricht beeinflusst.
- Die **Ausbreitungsverzögerung** hängt von der Signallaufzeit und der zurückgelegten Strecke in dem verwendeten Medium ab.
- Die **Verarbeitungsverzögerung** entspricht der Zeit, die der Sender und der Empfänger benötigen, um die Nachricht zu verarbeiten. Diese Zeit beinhaltet z.B. das Parsen der Header und die Überprüfung der Prüfsumme. Diese Zeit wird sowohl von der Geschwindigkeit des Prozessors und des Arbeitsspeichers beeinflusst als auch von der Art der Implementierung.
- Die **Wartezeit** in Warteschlangen beeinflusst die RTT zusätzlich zu der Verarbeitungsverzögerung, falls mehr Nachrichten pro Zeit gesendet werden sollen, als gesendet werden können.

Zu Beginn dieses Abschnitts wird in Abschnitt 6.5.1 zunächst ein Vergleich der Latenz des DPS-Protokolls mit der 6LoWPAN-Implementierung der iSense-Plattform durchgeführt. Im Anschluss folgt in Abschnitt 6.5.2 ein Vergleich mit den 6LoWPAN-Implementierungen für TinyOS und in Abschnitt 6.5.3 ein Vergleich mit der DPS-Implementierung der Wiselib.

6.5.1. Vergleich mit der iSense-6LoWPAN-Implementierung

Um die RTT des DPS-Protokolls mit dem der nativen IPv6-Implementierung zu vergleichen, wird das ICMP-Protokoll verwendet. Der Sender sendet hierbei ICMP-Echo-Request-Nachrichten an den Client und misst die Zeit bis zum Empfang der zugehörigen Echo-Reply-Nachricht. Dies entspricht der RTT. Für das Experiment wird derjenige Server als Sender ausgewählt, der über eine DPS-Verbindung mit dem Client verbunden ist. Der Sender sendet insgesamt 100 Echo-Request-Nachrichten an den DPS-Client und wiederholt diesen Vorgang mit unterschiedlichen Nachrichtengrößen zwischen 8 und 512 Byte in Schritten von 8 Byte. Auf diese Weise werden zwischen dem Server und dem Client insgesamt 256000 Echo-Request- und Echo-Reply-Nachrichten ausgetauscht. Dieser Vorgang wird für unterschiedliche Konfigurationen des DPS-Protokolls

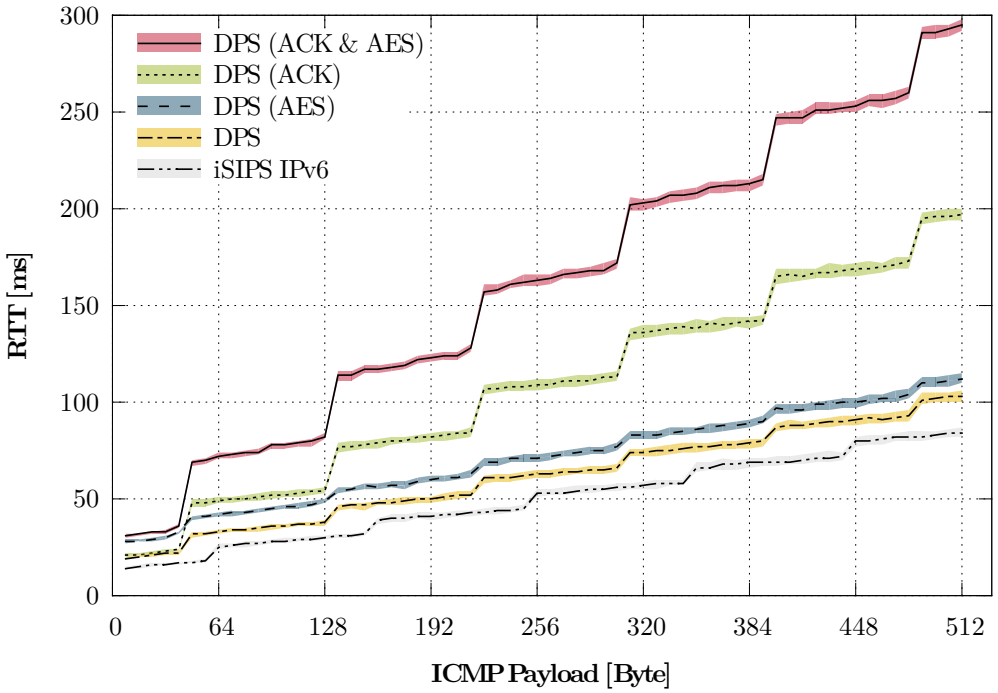


Abbildung 6.9.: Vergleich der Single-Hop-RTT zwischen der nativen IPv6-Implementierung mit 6LoWPAN und unterschiedlichen Konfigurationen des DPS-Protokolls

wiederholt (DPS-Protokoll mit/ohne Bestätigungen und mit/ohne AES-Prüfsummen). Anschließend wird dasselbe Experiment zwischen den beiden Servern durchgeführt, um die native IPv6-Implementierung mit dem DPS-Protokoll vergleichen zu können. Der direkte Vergleich der Kommunikation über das DPS-Protokoll zwischen Server und Client mit der Kommunikation über das 6LoWPAN/IPv6-Protokoll zwischen zwei Servern ermöglicht es, den Overhead des DPS-Protokolls direkt zu untersuchen.

Die Ergebnisse der Experimente sind in Abbildung 6.9 dargestellt, welche die RTT in Abhängigkeit von der Nachrichtengröße und dem verwendeten Protokoll zeigt. Hierbei geben die Kurven den Median der entsprechenden Messung an, während die farbigen Ränder den Bereich zwischen dem unteren und dem oberen Quartil angeben. In diesem farbigen Bereich befinden sich folglich 50 % aller Messwerte. Die Ergebnisse in Abbildung 6.9 zeigen, dass die RTT mit zunehmender Nachrichtengröße linear ansteigt, wobei die RTT zusätzlich in regelmäßigen Abständen sprunghaft ansteigt. Die RTT steigt linear mit zunehmender Nachrichtengröße an, da sich die Übertragungsverzögerung durch die zur Verfügung stehende konstante Bandbreite vergrößert. Die regelmäßigen sprunghaften Anstiege der RTT ergeben sich durch den verwendeten Fragmentierungsmechanismus: Die 802.15.4-Sicherungsschicht besitzt eine MTU von 127 Byte, so dass größere Nachrichten auf mehrere Frames verteilt werden müssen. In Abhängigkeit von der Länge des verwendeten Headers findet der erste Fragmentierungsschritt im Fall der nativen IPv6-Implementierung bei

	Einheit	iSIPS IPv6	DPS	DPS + AES	DPS + ACK	DPS + AES & ACK
m	$\frac{ms}{8\,Byte}$	0,5	0,6	0,7	0,6	1,3
f_s	$\frac{ms}{Segment}$	7	8	8	23	31
f_0	ms	14	19	28	20	31

Tabelle 6.1.: Ermittelte Werte der RTT-Parameter aus Abbildung 6.10

64 Byte ICMP-Payload statt, während dies beim DPS-Protokoll bereits bei 48 Byte der Fall ist. Dies liegt an dem zusätzlich vorhandenen DPS-Header, der Prüfsumme sowie der fehlenden 6LoWPAN-Schicht, die zur Kompression des IPv6-Headers verwendet wird. Des Weiteren fällt bei der Betrachtung der Ergebnisse auf, dass durch den Fragmentierungsmechanismus von 6LoWPAN anschließend alle 96 Byte ein zusätzliches Fragment erzeugt wird, während dies bei der Verwendung des DPS-Protokolls bereits alle 88 Byte geschieht. Dies liegt an der unterschiedlichen Größe der verwendeten Fragmentation-Header. Als Gegenmaßnahme für diesen Effekt wurden in Abschnitt 4.3.4 Mechanismen zur Kompression des DPS-Header, des DPS-Fragmentation-Header und des Nachrichteninhalts vorgestellt. Die zugehörige Evaluation dieser Mechanismen findet sich in Abschnitt 6.8.

Im Folgenden werden zunächst der lineare Anstieg innerhalb der Fragmente und anschließend der stufenhafte Anstieg durch das Versenden eines zusätzlichen Fragments für die unterschiedlichen Protokolle und Konfigurationen näher untersucht. Für die Beschreibung der Ergebnisse werden die Parameter, die in Abbildung 6.10 dargestellt sind verwendet: Hierbei gibt m die Zeit an, die zum Senden von acht zusätzlichen Byte benötigt wird, während f_s die Zeit angibt, die zum Senden eines zusätzlichen Fragments benötigt wird. Der Parameter f_0 steht getrennt hiervon für die Zeit, die zum Senden des ersten Fragments benötigt wird, da diese zusätzlich zu f_s noch die Verarbeitungszeit des gesamten Pakets vor der Fragmentierung beinhaltet. Die ermittelten Werte für die einzelnen Protokollkonfigurationen finden sich gesammelt in Tabelle 6.1 und werden im Folgenden genauer erläutert.

Bei der Verwendung des nativen IPv6-Protokolls erhöht sich die RTT um

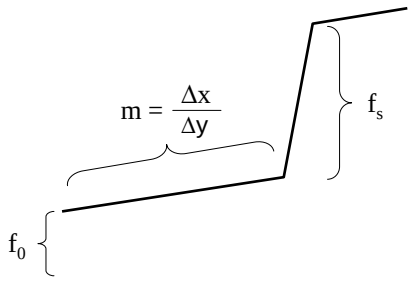


Abbildung 6.10.: Darstellung der Parameter, die zur Berechnung der Round-Trip-Time verwendet werden

$m = 0{,}5\,\text{ms} / 8\,\text{Byte}$. Dieser Anstieg erhöht sich auf 0,6 ms bei der Verwendung des DPS-Protokolls und auf 0,7 ms bei der Verwendung des DPS-Protokolls mit AES-Prüfsummen. Die Verwendung von Bestätigungen erhöht den linearen Anstieg der RTT nicht, da die Bestätigungen unabhängig von der Länge der Fragmente sind und pro Fragment verschickt werden. Werden hingegen Bestätigungen und AES-Prüfsummen gleichzeitig verwendet, so steigt die RTT um $m = 1{,}3\,\text{ms} / 8\,\text{Byte}$, da in diesem Fall die AES-Prüfsummen sowohl für die Fragmente als auch für die Bestätigungen berechnet werden müssen.

Der Parameter f_s, also die Zeit die zum Senden eines zusätzlichen Fragments benötigt wird, beträgt 7 ms bei der Verwendung des nativen IPv6-Protokolls. Dieser Wert erhöht sich auf 8 ms durch die Verwendung des DPS-Protokolls durch den zusätzlichen Verarbeitungsoverhead, jedoch unabhängig von der Verwendung der AES-Prüfsummen. Wird hingegen der Bestätigungsmechanismus verwendet, so erhöht sich f_s auf 23 ms, da in diesem Fall nach dem Senden jedes einzelnen Fragments auf die zugehörige Bestätigung des Kommunikationspartners gewartet werden muss, bevor das nächste Fragment verschickt werden kann. Die DPS-Implementierung sieht in diesem Fall vor, dass das nächste Fragment aus dem übergeordneten Paket erst dann erstellt wird, sobald das zugehörige Fragment erfolgreich verschickt wurde, also die Bestätigung angekommen ist. Dasselbe gilt für den Fall, dass sowohl Bestätigungen als auch AES-Prüfsummen verwendet werden, da in diesem Fall die AES-Prüfsumme für jedes Fragment und für jede Bestätigung zu genau dem Zeitpunkt berechnet wird, zu dem die Nachricht verschickt werden soll.

Bei dem letzten der drei untersuchten Parameter handelt es sich um f_0, der die Zeit darstellt, die zum Senden des ersten Fragments benötigt wird. Dieser Parameter beinhaltet den Overhead des IPv6- und 6LoWPAN-Protokolls, der zum Erstellen, Komprimieren und Interpretieren des entsprechenden Headers benötigt wird. Dieser beträgt 14 ms für das native IPv6-Protokoll und erhöht sich auf 19 ms durch das DPS-Protokoll. Zusammen mit den Daten für f_s lässt sich zusammenfassen, dass der Overhead des DPS-Protokolls 5 ms plus 1 ms/Fragment sowie 0,1 ms pro 8 Byte beträgt. Die Verwendung von AES-Prüfsummen erhöht zwar den Overhead pro Fragment nicht, benötigt jedoch zusätzliche 9 ms für das Senden und Empfangen der DPS-Nachrichten und weitere 0,1 ms für die Berechnung der Prüfsumme pro 8 Byte Payload.

Eine Zusammenfassende Darstellung der Daten aus Tabelle 6.1 und Abbildung 6.9 findet sich in Abbildung 6.11, welche die relative RTT der Konfigurationen des DPS-Protokolls in Bezug auf die native IPv6-Implementierung für Paketgrößen zwischen einem und sechs Fragmenten zeigt. Hierbei ist zu erkennen, dass der Overhead des DPS-Protokolls ohne Bestätigungen mit steigender Paketgröße asymptotisch abnimmt. Ohne AES-Prüfsummen sinkt der relative Overhead von +33 % auf +13 %, während er mit AES-Prüfsummen von +90 % auf +26 % sinkt. Dies wird durch zwei Faktoren bestimmt: Da das DPS-Protokoll ohne die Kompression des IPv6-Header durch die 6LoWPAN-Schicht arbeitet, ist der Overhead für kleinere IP-Pakete größer. Zusätzlich nimmt der Overhead durch die Berechnung der AES-Prüfsumme ab: Während die Berechnung

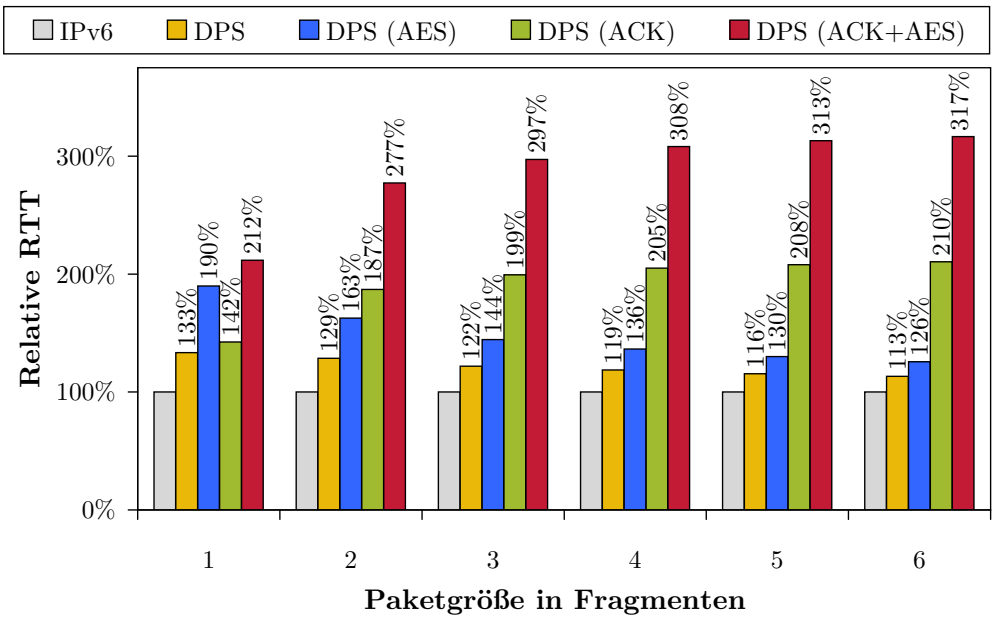


Abbildung 6.11.: Vergleich der RTT der iSIPS-IPv6-Implementierung mit den unterschiedlichen Konfigurationen des DPS-Protokolls (iSIPS-IPv6 = 100 %)

der Prüfsummen beim Versand einer Nachricht, die nur aus einem Fragment besteht, sequentiell erfolgt (Sender berechnet Prüfsumme, sendet Fragment. Empfänger empfängt Fragment, berechnet Prüfsumme), kann diese bei mehreren Fragmenten teilweise parallelisiert werden (Empfänger empfängt Fragment n und berechnet Prüfsumme n, gleichzeitig berechnet Sender Prüfsumme von Fragment n+1 und sendet Fragment n+1). Auf diese Weise fällt Parameter f_0 bei steigender Anzahl an Fragmenten weniger stark ins Gewicht. Bei der Verwendung von Bestätigungen hingegen steigt der Overhead von +42 % auf +110 % bzw. von +112 % auf +217 % bei der Verwendung von AES-Prüfsummen. Die Verdoppelung der RTT durch die Verwendung von Bestätigungen wurde bereits oben besprochen: Dies geschieht durch die Verdoppelung der versendeten Nachrichten, da zu jedem Fragment nun zusätzlich noch eine Bestätigung verschickt werden muss. In diesem Fall kann die Berechnung der AES-Prüfsumme nicht wie zuvor parallelisiert werden, da vor dem Versenden des nächsten Fragments die Bestätigung des vorangegangenen Fragments beim Sender angekommen sein muss.

Den größten Einfluss auf die RTT hat folglich die Verwendung von Bestätigungen. Da jedoch weder IP noch ICMP einen zuverlässigen Nachrichtentransport garantieren, sollte das DPS-Protokoll in diesen Fällen ohne Bestätigungen eingesetzt werden. Auf diese Weise wird die starke Erhöhung der RTT durch die Bestätigungen verhindert, ohne die Dienstgüte der Protokolle hinsichtlich ihrer Zuverlässigkeit einzuschränken, da diese Protokolle diesen Dienst ohnehin nicht bereitstellen. Dies gilt in selbem Maße für die Verwendung von UDP, weshalb die Experimente im nächsten Abschnitt auf die Verwendung von Bestätigungen ver-

zichten. Falls das TCP-Protokoll vom Client implementiert wird, kann ebenfalls auf Bestätigungen verzichtet werden, da TCP die notwendigen Mechanismen selbständig zur Verfügung stellt und auch im nativen IPv6 Fall den unzuverlässigen Dienst von IP nutzen würde. Auf die Verwendung von Bestätigungen sollte deshalb nur dann zurückgegriffen werden, wenn Protokollfunktionen ausgeführt werden, die den Zustand der beiden Kommunikationspartner verändern - ein Beispiel hierfür ist das Setzen der IP-Adresse des Clients durch den Server (siehe `setGlobalIpv6Address()` in Abschnitt 5.6).

Die Verwendung von AES-Prüfsummen hingegen erhöht die RTT nur in geringem Maße um einmalig 9 ms pro Paket und 0,1 ms/8 Byte Payload. Im Gegenzug wird die Integrität und Authentizität des Nachrichteninhalts aller Fragmente sichergestellt. Ob diese Funktionalität in dem Anwendungsszenario notwendig ist, in dem das DPS-Protokoll eingesetzt werden soll, muss der Anwendungsentwickler entscheiden.

Die Untersuchungen der RTT haben gezeigt, dass das DPS-Protokoll ohne die Verwendung von Bestätigungen und AES-Prüfsummen die RTT um lediglich 13% bis 33% erhöht, wobei der Overhead mit zunehmender Paketgröße abnimmt.

6.5.2. Vergleich mit anderen Implementierungen

Zum Vergleich werden an dieser Stelle zunächst die Ergebnisse von insgesamt drei externen Evaluationen herangezogen. In der Arbeit von [155] wird die b6LoWPAN-Implementierung (Stand: 7/2008) für TinyOS 2.1 auf Sensorknoten vom Typ TelosB untersucht. Eine Untersuchung des ebenfalls für TinyOS 2.1 verfügbaren BLIP (Stand: 12/2010) auf TelosB-Knoten findet sich in [156]. Die Autoren von [157] untersuchen die b6LoWPAN-Implementierung (Stand: 11/2008) für TinyOS 2.1 auf Sensorknoten vom Typ MicaZ und TelosB. Abbildung 6.12 fasst diese Ergebnisse zusammen und stellt die RTT über einen Hop dar, wobei jeweils der Mittelwert über 20 [157], 25 [155] oder 100 [156] Messungen angegeben ist. Die Ergebnisse des DPS- und iSIPS-Protokolls stellen wie bereits in Abschnitt 6.5 den Median aus 100 Messungen dar.

Wie in Abbildung 6.12 dargestellt, wächst die RTT für alle verglichenen Protokolle mit steigender Paketgröße. Hierbei fällt auf, dass die iSIPS-Implementierung auf der JN5148-Plattform im Vergleich zu den nativen IPv6-Implementierungen der anderen Hardware-Plattformen eine deutlich niedrigere RTT aufweist. Der direkte Vergleich zeigt eine Verbesserung um den Faktor 4 (b6LoWPAN, MicaZ), 6 (BLIP TelosB) bis zum Faktor 8 (b6LoWPAN TelosB). Die Verschlechterung der RTT durch das DPS-Protokoll fällt in jeder Konfiguration niedriger als dieser Faktor aus: Das DPS-Protokoll weist selbst unter Verwendung von Bestätigungen und AES-Prüfsummen eine niedrigere RTT als die beste Vergleichsplattform (b6LoWPAN auf MicaZ) auf. Zusammenfassend führt das DPS-Protokoll folglich zwar zu einer Verschlechterung der erreichten RTT der IPv6/6LoWPAN um etwa 30 % (ohne Bestätigungen), dies führt jedoch nicht zu einer Verschlechterung der RTT über die von den anderen Plattformen erreichten Werte.

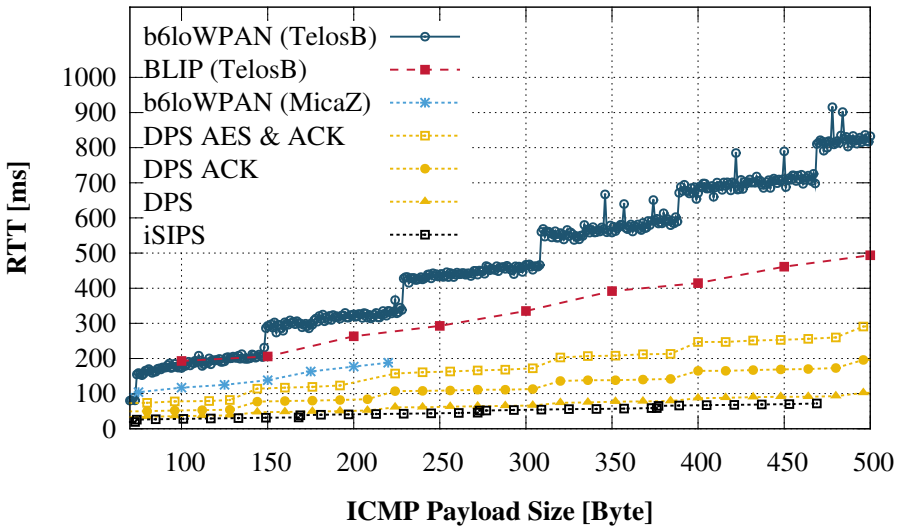


Abbildung 6.12.: Vergleich der Round-Trip-Time mit BLIP auf TelosB [156] sowie b6loWPAN auf TelosB [155, 157] und MicaZ [157]

6.5.3. Vergleich mit der Wiselib-Implementierung

In diesem Abschnitt wird ein Vergleich der Implementierung für die iSense-Plattform mit der Implementierung des DPS-Protokolls für die Wiselib durchgeführt. Ein direkter Vergleich der beiden Implementierungen findet sich in Abbildung 6.13, die die erzielte RTT in Abhängigkeit von der ICMP-Payload-Länge zeigt. Hierbei werden zusätzlich die 6LoWPAN-Implementierung der iSense-Plattform und der Wiselib miteinander verglichen, welche die Grundlage für die vom DPS-Protokoll verwendeten Server-Skeletons (siehe Abschnitt 5.6) bilden. Bereits bei diesem Vergleich der 6LoWPAN-Implementierungen ist zu erkennen, dass die Wiselib-Implementierung eine (geringfügig) niedrigere Round-Trip-Time aufweist als die iSense-Implementierung. Dieser Effekt ist noch deutlicher beim DPS-Protokoll sowie beim DPS-Protokoll mit Bestätigungen zu erkennen.

Um eine bessere Vergleichsmöglichkeit zu bieten, wurden aus diesen Diagrammen die in Abbildung 6.10 auf Seite 122 dargestellten Parameter ermittelt und in Tabelle 6.2 zusammengestellt. Aus den in dieser Tabelle dargestellten Daten kann abgelesen werden, wie sich die Round-Trip-Time der unterschiedlichen Implementierungen zusammensetzt: Die Zeit, die zum Senden eines Bytes benötigt wird (Parameter m) unterscheidet sich zwischen den beiden Plattformen nicht. Die Zeit hingegen, die vom DPS-Protokoll für das Senden eines Fragments verwendet wird (Parameter f_s), konnte bei der Wiselib-Implementierung um 25 % verringert werden. Die größte Änderung findet sich jedoch bei Parameter f_0, also der Zeit, die für das Senden des ersten Fragments benötigt wird. Dieser Parameter gibt die Dauer an, die sich aus dem Overhead des IPv6- und 6LoWPAN-Protokolls einerseits und dem Overhead des DPS-Protokolls andererseits zusammensetzt. Dies beinhaltet das Erstellen, Komprimieren und Interpretieren der entsprechenden Header und die Verarbeitung der Nachrichten. Diese Zeit konnte bereits beim 6LoWPAN-Protokoll von 14 auf 4 ms im

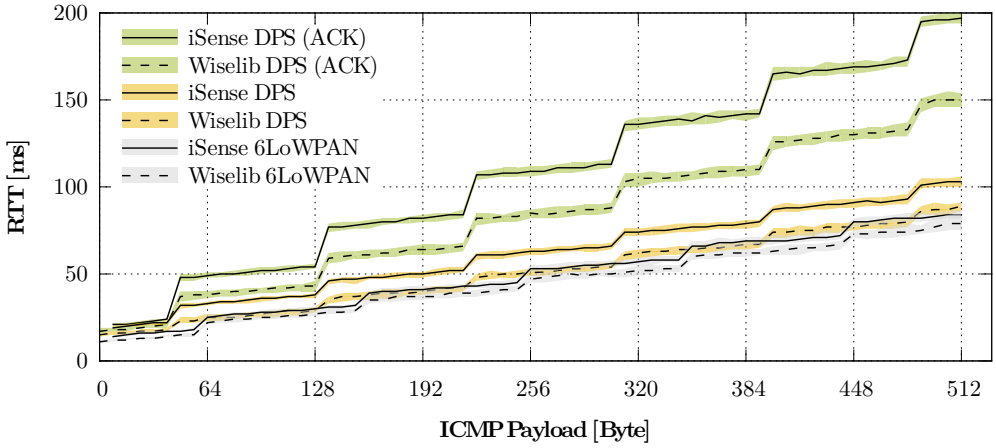


Abbildung 6.13.: Vergleich der ICMP-Round-Trip-Time der Implementierung des IPv6/6LoWPAN- und DPS-Protokolls für die iSense-Plattform mit der Implementierung für die Wiselib

Rahmen der Wiselib-Implementierung verbessert werden, wodurch auch die DPS-Implementierung eine um 10 ms bessere Round-Trip-Time aufweist.

Ein Grund für die niedrigere Round-Trip-Time der 6LoWPAN- und DPS-Implementierungen für die Wiselib ist die Einschränkung der Wiselib, keine dynamische Speicherverwaltung zu unterstützen. Dies bedeutet, dass sämtliche Puffer für Nachrichten nicht dynamisch zur Laufzeit mittels `malloc`[2] auf dem Heap erzeugt werden, sondern bereits bei der Initialisierung der Protokolle statisch angelegt werden oder aber zur Laufzeit auf dem Stack.

Durch den Verzicht auf dynamischen Speicher auf dem Heap wird die Geschwindigkeit der Wiselib-Implementierung erhöht, da das Anlegen von Speicher auf dem Heap langsamer ist als auf dem Stack: Um ein Objekt oder einen Puffer auf dem Stack anzulegen, muss lediglich der Stack-Pointer verschoben werden, während durch die dynamische Speicherverwaltung auf dem Heap beim Erstellen zunächst ein freier Speicherplatz gesucht werden muss. Andererseits können Elemente auf dem Stack weder wachsen noch schrumpfen, so dass immer so viel Speicher auf dem Stack reserviert werden muss, wie im Worst-Case benötigt wird. Falls weniger Speicher zur Laufzeit benötigt wird, wird dieser Speicher trotzdem reserviert. Aus diesem Grund ist der Speicherverbrauch der Wiselib-Implementierung wesentlich höher als die der iSense-Implementierung.

6.6. Datendurchsatz

Eine weitere wichtige Rolle bei der Evaluation der Leistung eines Netzwerkprotokolls stellt die Menge an Daten dar, die pro Zeiteinheit über ein Netzwerk von einem Sender zu einem Empfänger übertragen werden kann. Hierbei werden die Datenübertragungsrate und der Datendurchsatz unterschieden, die auch als

[2]Siehe: http://www.cplusplus.com/reference/cstdlib/malloc/

		iSense			Wiselib		
	Einheit	6LoWPAN	DPS	DPS+ ACK	6LoWPAN	DPS	DPS+ ACK
m	$\frac{ms}{8\,Byte}$	0,5	0,6	0,6	0,5	0,6	0,6
f_s	$\frac{ms}{Segment}$	7	8	23	7	6	17
f_0	ms	14	19	20	4	9	0

Tabelle 6.2.: Ermittelte Werte der RTT-Parameter aus Abbildung 6.10 für die iSense- und Wiselib-DPS-Implementierung (siehe Seite 122)

Brutto- und Nettorate bezeichnet werden. Die Datenübertragungsrate steht für die Menge an Daten, die innerhalb einer Zeiteinheit über einen Kanal übertragen werden kann, während der Datendurchsatz der Datenübertragungsrate abzüglich des Protokoll-Overheads entspricht. Während z.B. der von Sensornetzen häufig verwendete Funkstandard IEEE 802.15.4 eine maximale Datenübertragungsrate von 250 kbit/s (31,25 kB/s) besitzt, kann der erreichbare Datendurchsatz auf Anwendungsebene in Abhängigkeit von den verwendeten Protokollen niedriger ausfallen. Dies wird einerseits durch den Overhead der Protokoll-Header und Prüfsummen sowie andererseits von dem verwendeten Medienzugriffsverfahren oder Duty-Cycle-Protokoll beeinflusst.

In den nachfolgenden Experimenten wird das UDP-Protokoll verwendet, so dass der Protokoll-Overhead der UDP-, IPv6/6LoWPAN- und 802.15.4-Schicht berücksichtigt werden müssen: Das UDP-Protokoll besitzt einen Overhead von 8 Byte (6 Byte Header und 2 Byte Prüfsumme), während IPv6/6LoWPAN einen Header von 33 Byte besitzen. Die 802.15.4-Sicherungsschicht hingegen besitzt bei der Verwendung von 64-Bit MAC-Adressen einen Header von 23 Byte. Dies ergibt einen Gesamt-Overhead von 64 Byte, dem eine MTU von 127 Byte gegenüber steht, so dass im ersten Frame lediglich 63 Byte Nutzdaten versendet werden können. Zusätzlich beeinflusst das Medienzugriffsverfahren CSMA/CA den Datendurchsatz, da vor dem Senden jedes Frames das Clear-Channel-Assessment (CCA) durchgeführt werden muss. In der Arbeit von [158] wird für IEEE 802.15.4 unter Verwendung von 64-Bit Adressen ein maximaler theoretischer Datendurchsatz von 54,8 % (17,13 kB/s) des zur Verfügung stehenden Datenübertragungsrate angegeben, wobei 48,8 % (15,25 kB/s) experimentell unter Verwendung des Freescale MC13192 Funkchips nachgewiesen werden konnten. Dieser Wert berücksichtigt sowohl den Overhead der Sicherungsschicht von 23 Byte als auch das CCA des CSMA/CA-Verfahrens, beinhaltet jedoch nicht den Overhead des UDP sowie IPv6/6LoWPAN-Protokolls, wodurch sich der Datendurchsatz weiter verringert.

Zu Beginn dieses Abschnitts wird in Abschnitt 6.6.1 zunächst ein Vergleich des DPS-Protokolls mit der 6LoWPAN-Implementierung der iSense-Plattform durchgeführt. Im Anschluss folgt in Abschnitt 6.6.2 ein Vergleich mit den 6LoWPAN-Implementierungen für TinyOS sowie mit der DPS-Implementierung der Wiselib.

6.6.1. Vergleich mit der iSense-6LoWPAN-Implementierung

Um den Datendurchsatz des DPS-Protokolls mit der nativen IPv6-Implementierung vergleichen zu können, wurden wie bei allen Single-Hop-Experimenten die beiden DPS-Server und der DPS-Client in dem in Abschnitt 6.2 beschriebenen Versuchsaufbau verwendet. Zunächst wurde der Datendurchsatz zwischen den beiden DPS-Servern unter Verwendung der nativen IPv6-Implementierung ermittelt. Anschließend wurde dasselbe Experiment zwischen einem DPS-Server und dem DPS-Client wiederholt, um den Datendurchsatz des DPS-Protokolls zu ermitteln. Zur Messung des maximalen Datendurchsatzes wurde auf dem Empfänger die Anzahl der empfangenen Byte pro Sekunde ausgegeben und anschließend das Maximum gebildet. Der Sender erhöht während des Experiments schrittweise seine Sendegeschwindigkeit, indem er Pakete einer festgelegten Größe in immer kürzerer zeitlicher Abfolge sendet. Der Sender beginnt mit einer Paketgröße von 63 Byte und einem Sendeintervall von 100 ms. Nach dem Versenden von 1000 UDP-Paketen wird das Experiment unterbrochen, bevor der Sender mit einem Sendeintervall von 99 ms fortfährt. Sobald ein Sendeintervall von 0 ms erreicht wird, erhöht der Sender die Paketgröße auf 152 Byte und beginnt erneut mit einem Sendeintervall von 100 ms. Auf diese Weise wird sowohl der Einfluss der Sendegeschwindigkeit als auch der Paketgröße auf den erzielten Datendurchsatz ermittelt. Die festgelegten Paketgrößen entsprechen hierbei dem maximalen UDP-Payload, der in eine gegebene Anzahl Frames der Sicherungsschicht passt, d.h. 63 Byte entsprechen dem maximalen UDP-Payload in einem Frame, während 536 Byte dem maximalen Payload in sechs Frames entsprechen. Das Experiment wird anschließend unter Verwendung des DPS-Protokolls wiederholt, wobei die Paketgrößen entsprechend angepasst werden.

Die Ergebnisse dieses Experiments sind in Abbildung 6.14 dargestellt, die den maximalen Datendurchsatz der nativen IPv6-Implementierung und des DPS-Protokolls in Abhängigkeit von der Sendegeschwindigkeit und der Paketgröße zeigen. Aus den Ergebnissen geht hervor, dass der Datendurchsatz mit steigender Sendegeschwindigkeit linear ansteigt, bis er den maximalen Datendurchsatz erreicht, der von der gewählten Paketgröße abhängt. Eine weitere Erhöhung der Sendegeschwindigkeit hat keine Erhöhung des Datendurchsatzes zur Folge, führt jedoch zu einer Verlängerung der Sendewarteschlange, wodurch sich die Latenz (siehe Abschnitt 6.5) dieser Pakete vergrößert. In Abhängigkeit von der maximalen Länge der Sendewarteschlange kann es in diesem Fall zu einem Anstieg im Nachrichtenverlust kommen, da UDP-Pakete, die nicht in die Warteschlange eingereiht werden können, verworfen werden.

Die Ergebnisse in Abbildung 6.14 zeigen, dass sich das DPS-Protokoll und das IPv6/6LoWPAN-Protokoll lediglich in der Höhe des maximal erreichbaren UDP-Datendurchsatzes unterscheiden. Die maximalen Übertragungsraten sind in Abbildung 6.15 für die untersuchten Paketgrößen und die unterschiedlichen Konfigurationen des DPS-Protokolls aufgelistet. Während IPv6/6LoWPAN eine maximale Übertragungsrate von 12,5 kB/sec erreicht, erreicht das DPS-Protokoll maximal 12,0 kB/sec. Während die Verwendung von AES-Prüfsummen

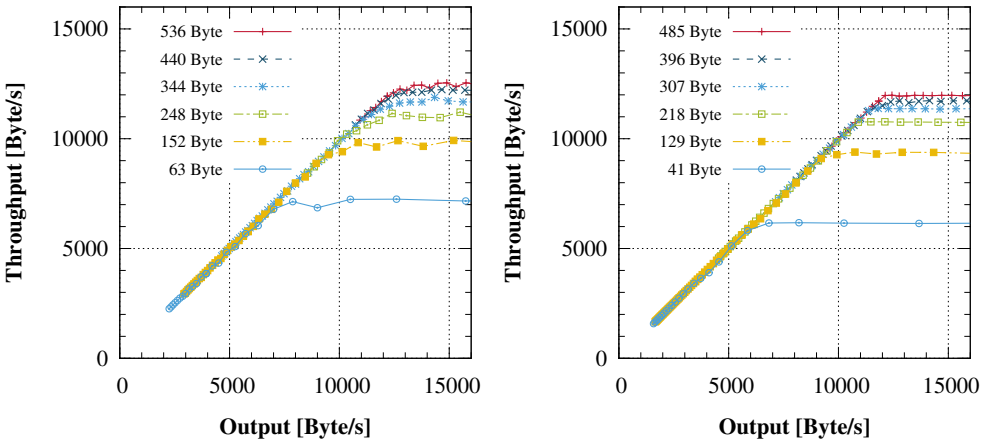


Abbildung 6.14.: Vergleich des Datendurchsatzes in Abhängigkeit von der Sendegeschwindigkeit und Paketgröße zwischen IPv6/6LoWPAN (a) und dem DPS-Protokoll (b)

bei Paketgrößen von ein bis zwei Fragmenten noch für einen Unterschied in der erreichten Übertragungsrate sorgt (etwa 0,8 kB/sec), beeinflusst die Verwendung von AES-Prüfsummen die Übertragungsrate bei größeren Paketen nicht mehr.

Dies liegt an der Verwendung des AES-Koprozessors, der die Berechnung der Prüfsummen hardwarebeschleunigt in derselben (oder einer höheren) Geschwindigkeit durchführt, mit der diese vom Funkchip gesendet werden können. Hierdurch erhöht sich zwar die Latenz des DPS-Protokoll (siehe Abschnitt 6.5), der Datendurchsatz verringert sich jedoch nicht. Sollte die Paketlänge nur ein Fragment betragen, so verringert sich der Datendurchsatz dadurch, dass der Sender und der Empfänger ihre Prüfsummen sequentiell berechnen müssen, während dies bei mehreren Nachrichten parallelisiert geschehen kann: Während der Sender die Prüfsumme für Fragment i berechnet, kann der Empfänger die Prüfsumme für Fragment i-1 berechnen.

Wie schon bei den Experimenten zur RTT in Abschnitt 6.5, beeinflussen Bestätigungen die Leistungsfähigkeit des DPS-Protokolls deutlich stärker als die Verwendung von AES-Prüfsummen und verringern den erreichten Datendurchsatz auf weniger als die Hälfte. Dies liegt daran, dass in diesem Fall die Fragmente des DPS-Protokolls nicht unmittelbar hintereinander versendet werden können, sondern jedes Mal auf den Empfang der Bestätigungen gewartet werden muss. Die hierfür benötigte Zeit entspricht der RTT zwischen Sender und Empfänger.

Durch die Verwendung von AES-Prüfsummen verringert sich der maximale Datendurchsatz in diesem Fall weiter, da keine Parallelisierung der Prüfsummenberechnungen wie oben beschrieben stattfindet, sondern die Prüfsummen weiterhin sequentiell von Sender und Empfänger berechnet werden.

Einen Vergleich des relativen Datendurchsatzes findet sich in Abbildung 6.16. In dieser Abbildung wird der maximale Datendurchsatz des IPv6/6LoWPAN-

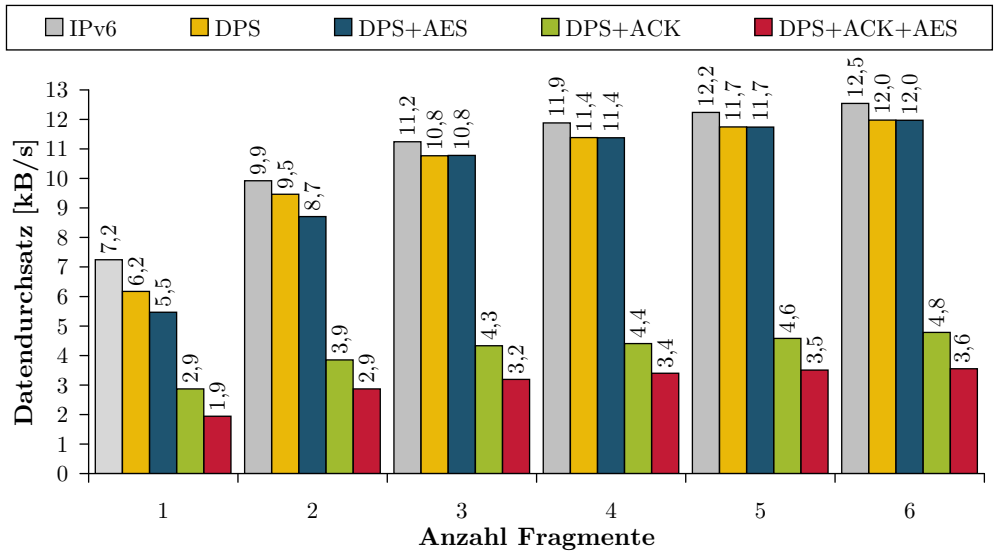


Abbildung 6.15.: Vergleich des absoluten Datendurchsatzes des IPv6-Protokolls mit unterschiedlichen Konfigurationen des DPS-Protokolls

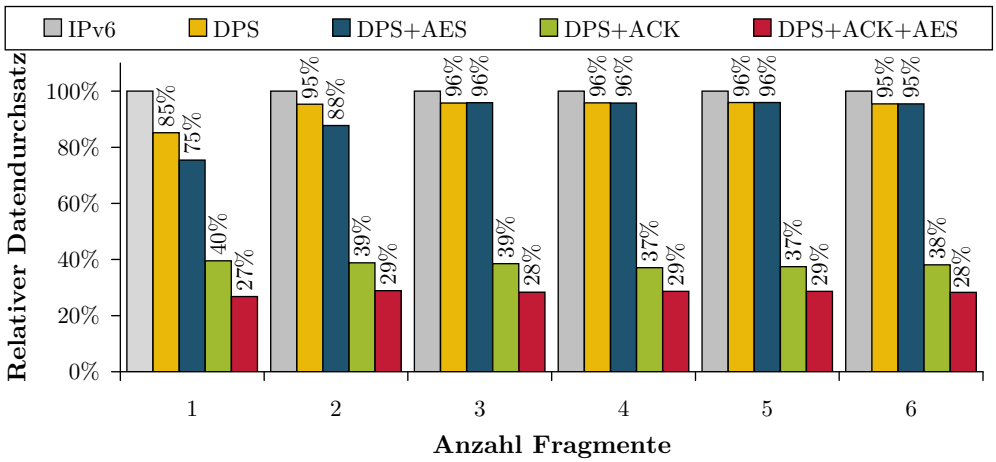


Abbildung 6.16.: Vergleich des relativen Datendurchsatzes des IPv6-Protokolls (100 %) mit unterschiedlichen Konfigurationen des DPS-Protokolls

Protokolls für die jeweilige Anzahl an Fragmenten als Vergleichsbasis für die unterschiedlichen Konfigurationen des DPS-Protokolls verwendet. Hierbei ist wie beim absoluten Vergleich zu erkennen, dass ab einer Paketgröße von drei oder mehr Fragmenten das DPS-Protokoll ohne Bestätigungen ein Maximum von 96 % des Datendurchsatzes des IPv6/6LoWPAN-Protokolls erreicht, während durch die Verwendung von Bestätigungen lediglich 38-40 % (28-29 % mit AES-Prüfsummen) erreicht werden. Der prozentual geringere Datendurchsatz für ein oder zwei Fragmente ist nicht nur auf den zusätzlichen Overhead durch die Header-Informationen und Verarbeitungszeit des DPS-Protokoll, sondern auch auf das Fehlen der Kompression des IPv6-Headers durch die 6LoWPAN-Schicht

zu erklären. Hierdurch verringert sich die maximal übertragbare Nutzlast im ersten Fragment von 63 Byte auf 41 Byte, wodurch sich der Protokoll-Overhead von 64 Byte um 35 % auf 86 Byte erhöht, so dass der Datendurchsatz um 25 % verringert wird.

6.6.2. Vergleich mit anderen Implementierungen

In diesem Abschnitt wird der Datendurchsatz des DPS-Protokolls mit zwei weiteren Plattformen verglichen: In der Arbeit von [159] wird der Datendurchsatz des BLIP-Protokoll unter TinyOS 2.1 unter Verwendung des Atmel RF230 und RF212 Funkchips untersucht, wie er unter anderem von Sensorknoten des Typs AVR-Raven verwendet wird. Zusätzlich wird wie bereits im vorherigen Abschnitt die Arbeit von [155] herangezogen, welche die b6LoWPAN-Implementierung für TinyOS 2.1 auf Sensorknoten vom Typ TelosB untersucht.

Die Ergebnisse sind in Abbildung 6.17 zusammengefasst und zeigen den maximalen UDP-Datendurchsatz über einen Hop. Wie aus der Abbildung hervorgeht, erreicht die IPv6/6LoWPAN-Implementierung für die iSense-Plattform mit 12,5 kB/s den höchsten Datendurchsatz. Dieser wird durch das DPS-Protokoll auf 12,0 kB/s reduziert (ohne Bestätigungen), während BLIP lediglich 10,6 bis 10,9 kB/s erreicht. Wird das DPS-Protokoll hingegen mit Bestätigungen verwendet, reduziert sich der Datendurchsatz auf 4,8 kB/s, was demselben Datendurchsatz entspricht, der von b6LoWPAN auf Sensorknoten vom Typ TelosB erreicht wird. Der hier dargestellte Vergleich zeigt, dass der Einsatz des DPS-Protokolls zwar für eine Verschlechterung des Datendurchsatzes sorgt, andererseits wird dennoch ein höherer Datendurchsatz erreicht, als dies bei BLIP oder b6LoWPAN der Fall ist. Wie bereits in Abschnitt 6.5 ausgeführt wurde, sollte auf den Einsatz von Bestätigungen nur genau dann zurückgegriffen werden, wenn durch eine Operation der Zustand der beteiligten Kommunikationspartner verändert wird oder das Protokoll auf eine zuverlässige Kommunikation angewiesen ist, ohne diese selbst bereit zu stellen. Auch die Ergebnisse in diesem und dem vorangegangenen Abschnitt unterstützten diese Feststellung.

Einen Vergleich des UDP-Datendurchsatzes der Implementierung des DPS-Protokolls für die iSense-Plattform mit der Implementierung für die Wiselib-Plattform findet sich in Abbildung 6.18. Im Gegensatz zu den im vorangegangenen Abschnitt vorgestellten Ergebnissen zur Round-Trip-Time unterscheidet sich der maximale Datendurchsatz der beiden Implementierungen nur geringfügig: Während der Datendurchsatz der Wiselib-Implementierung beim ersten Fragment 0,5 kB/s höher ausfällt als bei der iSense-Implementierung, jedoch ebenfalls 0,5 kB/s niedriger als die iSense-IPv6-Implementierung, verbessert sich dieser Sachverhalt für größere Nachrichten. Ab einer Nachrichtengröße von zwei Fragmenten ist der UDP-Datendurchsatz der Wiselib-Implementierung sogar 0,1 bis 0,2 kB/s höher als die iSense-IPv6-Implementierung und durchgehend 0,6 kB/s höher als die DPS-Implementierung für die iSense-Plattform.

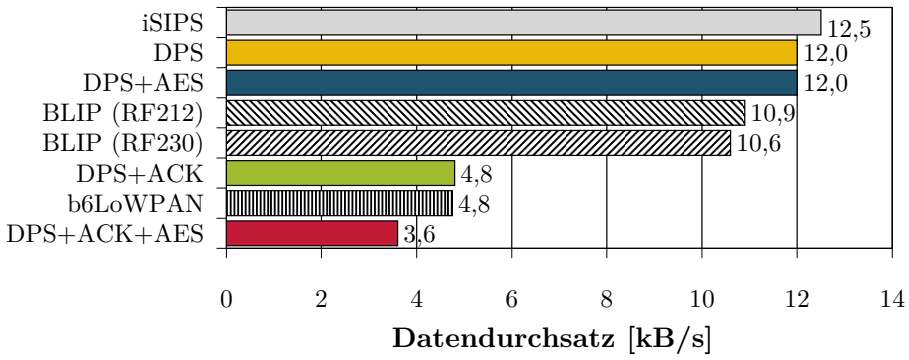


Abbildung 6.17.: Vergleich des Datendurchsatzes der nativen iSense-IPv6-Implementierung und der unterschiedlichen Konfigurationen des DPS-Protokolls mit BLIP [159] sowie b6loWPAN auf TelosB [155]

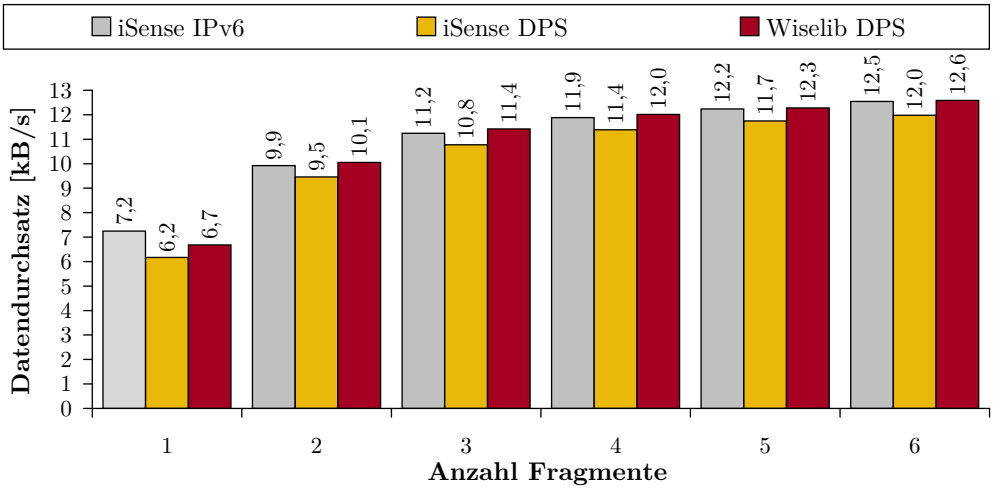


Abbildung 6.18.: Vergleich Datendurchsatzes der Implementierung des IPv6- und DPS-Protokolls für die iSense-Plattform mit der Implementierung für die Wiselib

6.7. Paketankunftsrate

Neben der Latenz und dem Datendurchsatz ist auch die zu erwartende Paketankunftsrate ein wichtiges Leistungsmerkmal eines Protokolls. Deshalb soll in diesem Abschnitt die Paketankunftsrate der nativen IPv6-Implementierung der iSense-Plattform mit unterschiedlichen Konfigurationen des DPS-Protokolls verglichen werden. Als Basis dieser Untersuchungen dienen die Paketankunftsraten, die im Rahmen der Latenz-Evaluation in Abschnitt 6.5 beim Versenden der ICMP-Echo-Request- und Echo-Response-Nachrichten ermittelt wurden. Die Ergebnisse sind in Abbildung 6.19 dargestellt, welche die Single-Hop-Paketankunftsrate im Abhängigkeit von der ICMP-Payload-Länge für das native IPv6-Protokoll und das DPS-Protokoll in insgesamt vier Konfigurationen (mit-/ohne Bestätigungen bzw. AES-Prüfsummen) darstellt. Die Payload-Länge

wurde auch hierbei in Schritten von 8 Byte bis zu einer maximalen Länge von 512 Byte erhöht, wobei jeweils 1000 ICMP-Echo-Request-Nachrichten von Server an den Client versendet wurden. Die Paketankunftsrate wurde anschließend aus dem Verhältnis zwischen gesendeten ICMP-Echo-Request-Paketen und empfangenen ICMP-Echo-Response-Paketen berechnet. Die dargestellte Paketankunftsrate stellt somit das Produkt aus der Paketankunftsrate des Senders und des Empfängers dar, da ein Echo-Response-Paket nur genau dann empfangen wurde, wenn sowohl das Echo-Request-Paket als auch das Echo-Response-Paket erfolgreich übertragen wurden.

Wie in Abbildung 6.19 zu sehen ist, ist die Paketankunftsrate unabhängig von der ICMP-Payload-Länge und unabhängig von dem gewählten Protokoll sehr hoch und beträgt im schlechtesten Fall 99,4 %. Sowohl die native IPv6-Implementierung, als auch das DPS-Protokoll verwenden hierbei die Bestätigungen der Sicherungsschicht, die jeden Frame bis zu drei Mal sendet, falls keine Bestätigung empfangen wurde. In den Konfigurationen des DPS-Protokoll mit eigenem Bestätigungsmechanismus (DPS+ACK und DPS+ACK+AES in Abbildung 6.19) werden keine Bestätigungen der Sicherungsschicht verwendet und stattdessen lediglich der Bestätigungsmechanismus des DPS-Protokolls. Die Paketankunftsrate während der gesamten Dauer des Experiments beträgt in diesen Fällen 100 %.

Dies zeigt, dass das DPS-Protokoll keinen negativen Einfluss auf die zu erwartende Paketankunftsrate hat und dass der Bestätigungsmechanismus des DPS-Protokoll zu einer höheren Paketankunftsrate führt als die Bestätigungen der Sicherungsschicht. Insgesamt konnte in diesem Experiment eine sehr hohe Paketankunftsrate nachgewiesen werden. Dies liegt unter anderem an dem verwendeten Versuchsaufbau, bei dem kein zusätzlicher Nachrichtenverkehr im Hintergrund existierte und zeigt die korrekte Funktionsweise des DPS-Protokolls. Einen Vergleich der Paketankunftsrate unter realistischeren Randbedingungen findet sich in Abschnitt 7.3.3.

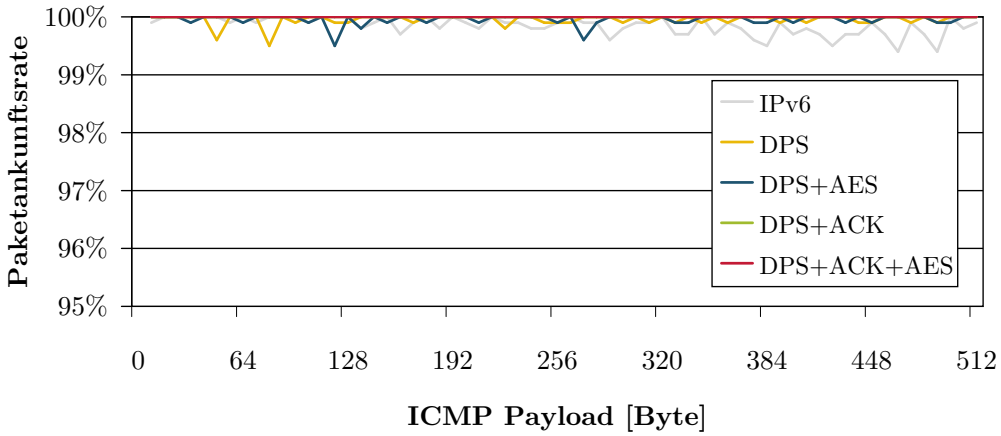


Abbildung 6.19.: Vergleich der Paketankunftsrate zwischen der nativen IPv6-Implementierung mit 6LoWPAN und unterschiedlichen Konfigurationen des DPS-Protokolls

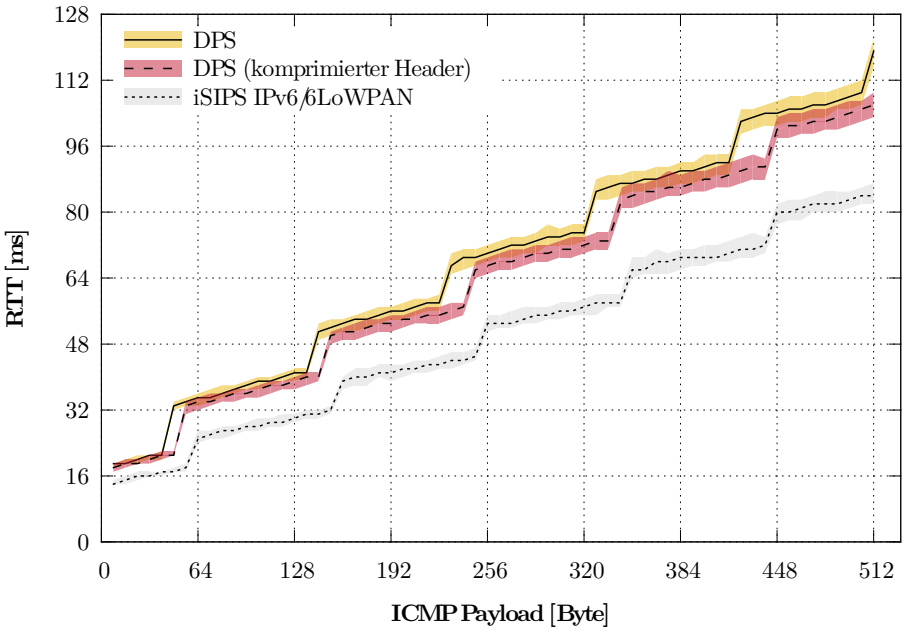


Abbildung 6.20.: Verbesserung der RTT durch Kompression der DPS-Header

6.8. Erweiterungen

Um die Leistung des DPS-Protokolls zu verbessern, wurden in Abschnitt 4.3.4 zwei Optimierungsmechanismen vorgestellt: Die Kompression des DPS-Header sowie die Kompression des Nachrichteninhalts. Diese Mechanismen werden in den beiden nachfolgenden Abschnitten evaluiert. Die Evaluation verwendet hierbei dasselbe Szenario wie die restlichen Single-Hop-Experimente in diesem Kapitel.

6.8.1. Kompression des DPS-Headers

Wie in Abschnitt 4.2 beschrieben wurde, kann durch die Verwendung von zustandsloser sowie zustandsbehafteter Kompression die Größe des DPS-Headers von 11 Byte auf 5 Byte reduziert werden. Zusätzlich kann der Fragmentation-Header von 4 Byte auf 2 Byte verkleinert werden. Auf diese Weise ist es möglich, den Overhead des DPS-Protokoll zu verringern und die Fragmentierungslänge auf denselben Wert wie bei der nativen IPv6-Implementierung zu reduzieren.

Ein Vergleich des komprimierten und unkomprimierten DPS-Protokoll findet sich in Abbildung 6.20, welche die RTT in Abhängigkeit von der Paketgröße darstellt. In den Ergebnissen ist zu erkennen, dass durch die Kompression der DPS-Header die RTT bei Einsatz des DPS-Protokolls innerhalb der Fragmente um 4 % reduziert werden konnte. Zusätzlich sind durch die Vergrößerung der Nutzlast durchschnittlich weniger Fragmente notwendig, um dieselbe Datenmenge zu transportieren. In einigen Fällen ist hierdurch eine Reduktion der RTT durch die Einsparung eines Fragments um 14 % möglich.

6.8.2. Kompression des Nachrichteninhalts

Für die Kompression des Nachrichteninhalts wird der in [125] vorgestellte S-LZW-Algorithmus verwendet. Die Portierung dieses Algorithmus für die iSense-Plattform basiert auf der Implementierung von S-LZW für den MSP430[3] und benötigt 2,6 kB Programmspeicher sowohl für den Server als auch den Client. Darüber hinaus benötigt der S-LZW-Algorithmus zur Kompression von n Byte zusätzliche $1{,}25 * n$ Byte Arbeitsspeicher, da der Algorithmus bei der Kompression der Daten nicht zwangsläufig kleinere Ausgabedaten produziert und im Worst-Case eine Vergrößerung der Daten um den Faktor 1,25 erzeugt.

Zur Evaluation des DPS-Protokoll unter der Verwendung von S-LZW wurden Daten verwendet, die auf den in [125] vorgestellten Kompressionsraten für die Datensätze aus dem Calgary-Corpus [160, 161], ZebraNet [162] sowie Great-Duck-Island [163] basieren. Der Calgary-Corpus stellt eine Sammlung von Text- und Binärdaten dar, die häufig zur Evaluation von Kompressionsalgorithmen verwendet werden. ZebraNet und Great-Duck-Island hingegen sind zwei bekannte Anwendungsszenarien für drahtlose Sensornetze.

Aus den in [125] gegebenen Kompressionsraten wurden synthetische Datensätze erzeugt, die vergleichbare Kompressionsraten für beliebige Längen aufweisen. Der Programmcode, der zur Erzeugung dieser Daten verwendet wurde, ist in Abbildung 6.21 gegeben, während die resultierenden Kompressionsraten in

[3] Verfügbar unter: https://sites.google.com/site/cmsadler/

```
1  for (uint16 i=0; i<ping_data_len_; i++) {
2   data[i] = 1;
3  }
4
5  if (selection=PING_LZW_TEST_DATA_CALGARY_CORPUS) {
6   for (uint16 i=0; i<ping_data_len_; i++) {
7    if (((i%50) > 10) && ((i%50) < 27)) {
8     data[i] = 0;
9    }
10  }
11 } else if (selection=PING_LZW_TEST_DATA_ZEBRA_NET) {
12  for (uint16 i=0; i<ping_data_len_; i++) {
13   if (((i%50) > 10) && ((i%50) < 34)) {
14    data[i] = 0;
15   }
16  }
17 } else if (selection=PING_LZW_TEST_DATA_GREAT_DUCK_ISLAND) {
18  for (uint16 i=0; i<ping_data_len_; i++) {
19   if (((i%50) > 10) && ((i%50) < 39)) {
20    data[i] = 0;
21   }
22  }
23 }
```

Abbildung 6.21.: Erzeugung der Eingabedaten für die Evaluation der Nachrichtenkompression

	Kompressionsrate				
Datensatz	Median	Minimum	Mittelwert	Maximum	[125]
Calgary Corpus	1,10	1,00	1,10	1,26	1,11
ZebraNet	1,37	1,16	1,36	1,60	1,38
Great Duck Island	1,60	1,26	1,59	1,82	1,59

Tabelle 6.3.: Resultierende Kompressionsraten der erzeugten Eingabedaten aus Abbildung 6.21 im Vergleich zu der Kompressionrate in [125]

Tabelle 6.3 dargestellt sind. Hierbei ist zu beachten, dass die resultierenden Kompressionsdaten nicht immer dem Mittelwert entsprechen, sondern innerhalb der angegebenen Minima und Maxima schwanken.

Die erzeugten Daten wurden als Payload für den Versand von ICMP-Paketen verwendet, um einen Vergleich zu den RTT-Experimenten in Abschnitt 6.5 zu ermöglichen. Das DPS-Protokoll wurde so angepasst, dass alle Eingabedaten vor dem Versand komprimiert werden, diese jedoch nur dann als Payload anstelle der Originaldaten verwendet werden, wenn die Länge der komprimierten Daten kleiner ist. Die Ergebnisse der Experimente sind in Abbildung 6.22 dargestellt, welche die RTT in Abhängigkeit von der steigenden ICMP-Payload-Länge zeigen. Aus den Ergebnissen geht hervor, dass die Daten aus dem Calgary-Corpus mit einer Kompressionsrate von 1,11 in 95 % der Fälle eine um durchschnittlich 15 % höhere RTT aufweist als das DPS-Protokoll ohne Kompression. Die Daten aus ZebraNet, die eine Kompressionsrate von 1,38 aufweisen, zeigen in 60 % der Fälle eine Verbesserung der RTT um durchschnittlich 3 %, während die RTT in den restlichen 40 % um durchschnittlich 13 % erhöht wird. Bei einem Kompressionsgrad von 1,59, wie bei den Daten von Great-Duck-Island, wird die RTT in 75 % aller Fälle um durchschnittlich 9 % (maximal 15 %) verbessert und nur in 25 % der Fälle um durchschnittlich 14 % verschlechtert.

Die Ergebnisse zeigen, dass der Einsatz der S-LZW-Kompression nur dann in einer Verbesserung der RTT resultiert, wenn eine Kompressionsrate von mehr als 1,38 erreicht werden kann, da erst bei den Daten mit einer Kompressionsrate von 1,59 eine überwiegende Verbesserung nachgewiesen werden konnte. Da die Kompressionsrate der Daten vor der Kompression nicht bekannt ist und aufgrund der unbekannten Struktur der Daten des entsprechenden Kommunikationsprotokolls nicht geschätzt werden kann, ist der Einsatz der S-LZW-Kompression nicht zu empfehlen. Der zusätzliche Overhead von 2,6 kB Programmspeicher und die Erhöhung des Speicherverbrauchs um das 1,25 fache der Payload-Länge wiegen die nur geringe Verbesserung der RTT um durchschnittlich 9 % und bis zu 15 % nicht auf. Stattdessen empfiehlt der Autor, das Kompressionsverfahren bereits auf Anwendungsebene zu verwenden, falls der Anwendungsprogrammierer bei der Konzeption des Sensornetzes eine entsprechende Kompressionsrate der Daten feststellt. Dies bietet zusätzlich die Möglichkeit, die Daten durch eine passende Transformation (z.B. Umsortierung) für die Kompression vorzubereiten.

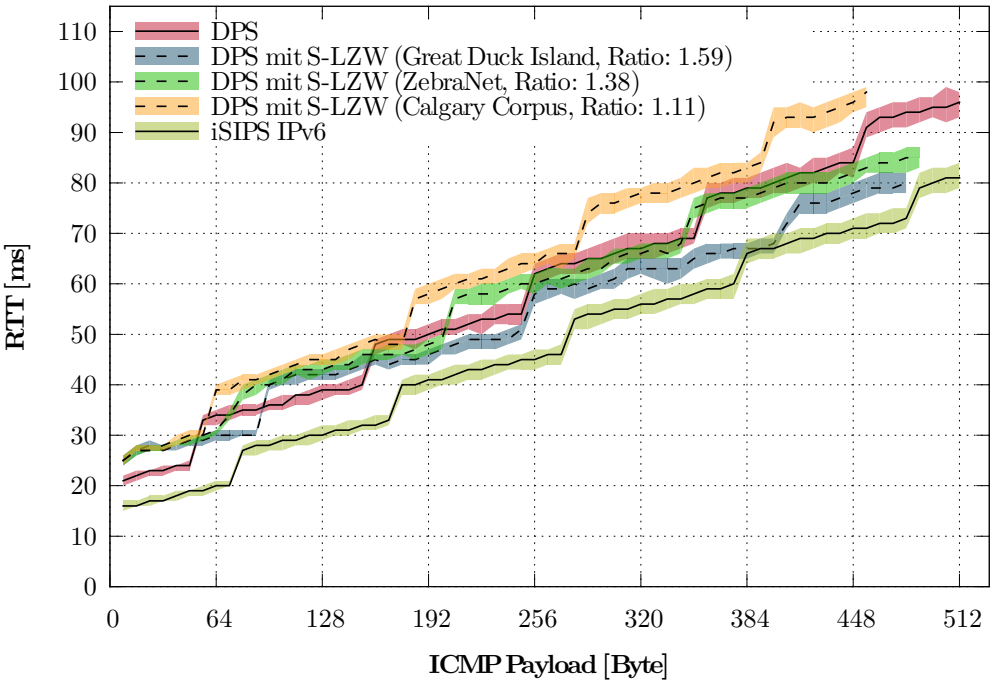


Abbildung 6.22.: Auswirkungen der Kompression des Nachrichteninhalts mittels S-LZW auf die RTT in Abhängigkeit von der Payload-Länge

6.9. Zusammenfassung

Die in diesem Kapitel durchgeführte Evaluation des DPS-Protokolls zeigt einerseits die Funktionsfähigkeit des Protokolls und bietet andererseits einen Überblick über die Auswirkungen des DPS-Protokolls auf die Kommunikation im Sensornetz in einem Single-Hop-Szenario. Jeder der in diesem Kapitel durchgeführten Versuche beinhaltet deshalb den direkten Vergleich mit der nativen IPv6-Implementierung, um die relativen Auswirkungen des DPS-Protokolls zu veranschaulichen. Zur Durchführung der Experimente wurde ein Versuchsaufbau unter Laborbedingungen gewählt, um die Ergebnisse der Evaluation nicht durch den zusätzlichen Nachrichtenaustausch eines Anwendungsprotokolls oder zwischen benachbarten Knoten zu beeinflussen und stattdessen die Leistung unter möglichst idealen Bedingungen zu vergleichen. Darüber hinaus beinhaltet dieses Kapitel den Vergleich zu den Implementierungen der IPv6- und 6LoWPAN-Protokolle auf anderen Hard- und Softwareplattformen, um eine allgemeine Vergleichsmöglichkeit zu bieten.

Im Rahmen der in diesem Abschnitt durchgeführten Evaluationen konnte gezeigt werden, dass die Implementierung des DPS-Protokolls und des zugehörigen IPv6-Stub und -Skeleton möglich ist und dass hierdurch die Programmgröße so weit reduziert werden konnte, dass auch eine Anbindung von Plattformen mit geringerem Programmspeicher über IPv6 möglich ist. Dies zeigt die Funktionsfähigkeit der Implementierung und die Anwendbarkeit des Konzepts der verteilten Protokollstapel. Durch den Einsatz des DPS-Protokolls ist der Betrieb

einer Anwendung, die IPv6 zum Nachrichtenaustausch nutzt, auf Plattformen mit geringerer Programmspeicherausstattung möglich. Dies ist ohne den Einsatz des DPS-Protokolls mit der nativen IPv6-Implementierung nicht möglich: Wie in Abschnitt 6.3 beschrieben wurde, konnte die Programmgröße auf dem Client um insgesamt stark verringert werden. Die Programmgröße auf dem Server hingegen stieg nur leicht durch die Integration des DPS-Protokolls. Neben der Programmgröße konnte auch der Speicherverbrauch auf dem Client um stark verringert werden, wohingegen sich der Speicherverbrauch auf dem Server nur geringfügig erhöht.

Zwar erlaubt das DPS-Protokoll den Einsatz von IPv6 auf Sensorknoten mit stärkeren Ressourcenbeschränkungen - das DPS-Protokoll kann jedoch auch negative Auswirkungen auf die Leistung der Kommunikation im Sensornetz haben. Den größten negativen Einfluss hat es auf die Round-Trip-Time: Im Rahmen der Experimente in Abschnitt 6.5 konnte hierbei gezeigt werden, dass die Round-Trip-Time durch das DPS-Protokoll um ca. 13 % bis 33 % in Abhängigkeit von der Größe der zu versendenden Nachricht erhöht wird. Durch den Einsatz von AES-Prüfsummen erhöht sich die RTT um weitere 13 bis 57 %, gleichzeitig wird jedoch die Sicherheit durch den Schutz der Integrität und Authentizität erhöht.

Der UDP-Datendurchsatz hingegen verringert sich für Nachrichten, die größer als ein Fragment sind, durch den Einsatz des DPS-Protokolls nur um 5 %. Werden Nachrichten mit AES-Prüfsummen verwendet, so reduziert sich der Datendurchsatz ab einer Nachrichtengröße von drei Fragmenten ebenfalls nur um 5 %. Auch der Einfluss des DPS-Protokolls auf die Paketankunftsrate ist nur sehr gering. Wie in Abschnitt 6.7 beschrieben wurde, verringert das DPS-Protokoll die Paketankunftsrate im schlimmsten Fall um lediglich 0,6 % auf 99,4 %.

Bei der Auswahl der vorgestellten Erweiterungen für das DPS-Protokoll muss auf die Anforderungen der jeweiligen Anwendung bzw. des verwendeten Protokolls geachtet werden. AES-Prüfsummen sollten genau dann verwendet werden, wenn ein Angreifer existiert, der die Kommunikation zwischen den Sensorknoten manipulieren könnte und keine anderen Mechanismen zum Schutz der Kommunikation (wie z.B. Verschlüsselung und Authentifizierung auf Sicherungsschicht) vorhanden sind oder diese nicht ausreichen. Bestätigungen hingegen sollten nicht für den Nachrichtenaustausch zwischen den Sensorknoten verwendet werden, sondern nur für wenige Funktionsaufrufe zwischen dem Client-Stub und Server-Skeleton, die den Zustand der Verbindung bzw. der Protokolle verändern (z.B. `getGlobalIpv6AddressFromServer()` beim IPv6-Protokoll, siehe Abschnitt 5.6).

Auf die Kompression des Nachrichteninhalts mittels S-LZW sollte verzichtet werden, da hierdurch nur in Ausnahmefällen eine Verbesserung der Latenz erreicht werden kann. Die S-LZW-Kompression führt in den meisten Fällen zu einer Verschlechterung der Latenz durch die Dauer der Kompression in Kombination mit den erreichbaren Kompressionsraten. Die Testdaten in Abschnitt 6.8.2 haben gezeigt, dass nur bei den Daten aus dem Great-Duck-Island-Szenario und einer

Kompressionsrate von 1,59 eine Verringerung der Latenz um durchschnittlich 9 % möglich ist.

Die Kompression des DPS-Nachrichten-Headers hingegen sollte in jedem Fall verwendet werden, da hierdurch die Latenz in allen Fällen im Vergleich zum DPS-Protokoll ohne Header-Kompression verringert werden kann. Darüber hinaus kann durch die Header-Kompression die Fragmentlänge angepasst werden, so dass genauso viele Fragmente über das DPS-Protokoll wie über die native IPv6/6LoWPAN-Implementierung verschickt werden müssen.

Zusammenfassend führt das DPS-Protokoll folglich zwar zu einer Verschlechterung der erreichten RTT, dies führt jedoch in keinem Fall zu einer Verschlechterung der RTT über die von den anderen Hard- und Softwareplattformen erreichten Werte. Auch für den Datendurchsatz konnte gezeigt werden, dass der Einsatz des DPS-Protokolls zwar für eine Verschlechterung des Datendurchsatzes sorgt, trotzdem wird auch in diesem Fall ein höherer Datendurchsatz erreicht, als dies bei BLIP oder b6LoWPAN der Fall ist.

Nachdem in diesem Abschnitt die Auswirkungen des DPS-Protokolls auf die Leistung der Single-Hop-Kommunikation unter Laborbedingungen untersucht worden sind, soll das DPS-Protokoll im nächsten Kapitel in einem Multi-Hop-Szenario untersucht werden. Die Single-Hop-Evaluation in diesem Kapitel wurde unter kontrollierten Bedingungen durchgeführt: Das Sensornetz beinhaltete lediglich drei Sensorknoten und neben der für die Durchführung der Experimente notwendigen Kommunikation wurden keine weiteren Nachrichten verschickt, die das Ergebnis beeinflussen könnten. Um das DPS-Protokoll in einem Anwendungsszenario zu testen, wird im nächsten Kapitel der Einsatz des DPS-Protokolls in einer Beispielanwendung aus dem Gebiet der Gebäudeüberwachung beschrieben. Die im Rahmen dieser Beispielanwendung gesammelten Daten sollen für eine Evaluation des DPS-Protokolls unter realen Bedingungen verwendet werden: Einerseits um die Eignung des DPS-Protokolls für solche Zwecke zu untersuchen und andererseits um die Auswirkungen auf die Kommunikation in einem Multi-Hop-Szenario mit einer Vielzahl von parallelen Datenströmen zu untersuchen.

7. Anwendungsfall: Gebäudeüberwachung

In diesem Kapitel wird die Evaluation des DPS-Protokolls anhand eines Anwendungsfalls durchgeführt. Während die Evaluation im vorangegangenen Kapitel 6 zur Untersuchung der Auswirkungen des DPS-Protokolls auf die Single-Hop-Kommunikation diente, soll in diesem Kapitel festgestellt werden, wie stark die Auswirkung der in Kapitel 6 gemessenen Effekte in einem Szenario mit Multi-Hop-Kommunikation sind. Darüber hinaus soll gezeigt werden, ob sich das DPS-Protokoll für den Einsatz in einer Sensornetzanwendung unter realen Bedingungen eignet. Ein Teil der in diesem Kapitel vorgestellten Ergebnisse wurde auf der CPScom 2013 in Peking [105] veröffentlicht.

Zu diesem Zweck wurde eine Anwendung zur Gebäudeüberwachung konzipiert, bei der die Sensorknoten ihre Messwerte in regelmäßigen Abständen an eine Basisstation weiterleiten. An dieser Basisstation werden sämtliche gesammelten Messwerte in einer Datenbank gespeichert und können über ein Webinterface abgerufen werden. In diesem Kapitel wird zunächst in Abschnitt 7.1 die Motivation hinter diesem Anwendungsfall beschrieben und ein Überblick über einige verwandte Arbeiten gegeben. Es folgt eine Beschreibung der Bestandteile des Demonstratorsystems in Abschnitt 7.2, gefolgt von der Evaluation des DPS-Protokolls in Abschnitt 7.3.

7.1. Motivation und verwandte Arbeiten

Wie bereits in Abschnitt 2.1.3 beschrieben wurde, eignen sich drahtlose Sensornetze sehr gut für den Anwendungsfall der Gebäudeüberwachung, da die Sensorknoten auch nachträglich in einem Gebäude eingebaut werden können und flexibel dort eingesetzt werden können, wo sie benötigt werden, ohne dafür Veränderungen an der Bausubstanz vornehmen zu müssen. Dies wird durch die drahtlose Kommunikation und die unabhängige Energieversorgung durch Batterien ermöglicht. In Abhängigkeit von dem gewünschten Einsatzzweck können Sensorknoten mit Sensoren für Temperatur, Luftfeuchtigkeit, Helligkeit oder Bewegungsmeldern ausgestattet werden. Auf diese Weise können Sensornetze z.B. zur Senkung der Heiz- und Energiekosten eines Gebäudes eingesetzt werden, indem die von den Sensorknoten gesammelten Messwerte als Datenquelle zur bedarfsgesteuerten Regelung der Beleuchtung oder Heizung eingesetzt werden. Zusätzlich können die Sensorknoten auch zur Überwachung von gesetzlichen Richtlinien eingesetzt werden, indem z.B. die Temperatur von Kühl- oder

Lagerräumen, die Lautstärke und Helligkeit in Büroräumen oder der Schadstoffgehalt in der Luft überwacht wird. Für diese Anwendungsfälle ist es wichtig, den jeweils aktuellen Messwert aller Sensoren zu kennen, um auf eine unerwünschte Veränderung der Messwerte reagieren zu können. Es ist oftmals jedoch auch notwendig, den zeitlichen Verlauf der Daten aufzuzeichnen, um z.B. periodische Veränderungen erkennen zu können oder um die Einhaltung bestimmter Richtlinien belegen zu können.

Die Anwendung von drahtlosen Sensornetzen zur Gebäudeüberwachung wurde in der Fachliteratur bereits mehrfach untersucht. So beschäftigt sich die Arbeit von [164] mit der Planung von WSNs zur Gebäudeüberwachung. Bei der Planung eines solchen Sensornetzes müssen zum einen alle zu überwachenden Messpunkte von mindestens einem Sensorknoten überwacht werden, ein weiterer wesentlicher Punkt ist jedoch auch die Konnektivität der Sensorknoten, so dass die Messwerte jedes Sensorknotens zu einer Datensenke transportiert werden können. Um diese beiden Anforderungen zu erfüllen, schlagen die Autoren aus [164] vor, zunächst die zur Überwachung notwendigen Sensorknoten an den gewünschten Positionen anzubringen, wobei z.B. darauf geachtet werden muss, dass ein Temperatursensor zur Messung der Raumtemperatur nicht direkt über einer Heizung angebracht wird. Anschließend werden dann Router-Knoten nach einer Two-Tiered-Relay-Knoten-Strategie (siehe Abschnitt 6.1.1) ausgebracht, die für die Konnektivität der Sensoren zur Datensenke verwendet werden.

Eine Arbeitsgruppe um David Culler von der UC Berkeley beschreibt in [165], wie ein Sensornetz zur Stromverbrauchs- und Helligkeitsmessung in Gebäuden eingesetzt werden kann. Die Autoren erläutern, dass bereits durch die Visualisierung des Stromverbrauchs der Energieverbrauch in einem Gebäude um 5 % bis 20 % gesenkt werden kann [166, 167]. Das von ihnen vorgestellte Sensornetz besteht aus insgesamt 44 Sensorknoten, wovon 38 zur Stromverbrauchsmessung und 6 zur Helligkeitsmessung verwendet werden. Alle Sensorknoten senden ihre aktuellen Messwerte einmal pro Minute mittels UDP an die Datensenke. Durch die Platzierung der Stromverbrauchsmesser an einzelnen Geräten sowie an den Mehrfachsteckdosen ist es den Autoren gelungen, den Stromverbrauch jedes angeschlossenen Geräts aus den gesammelten Daten zu berechnen.

Ein weiteres Beispiel für ein drahtloses Sensornetz zur Gebäudeüberwachung ist in [168] zu finden, in dem Bewegungsmelder und Helligkeitssensoren verwendet werden, um die Beleuchtung in einem Gebäude an die Anwesenheit von Personen und die Umgebungshelligkeit anzupassen. In Kombination mit Sensoren, die den Zustand der Lampen im Raum erfassen (ein- oder ausgeschaltet), versuchen die Autoren, die Beleuchtung an die Tageszeit und Sonneneinstrahlung anzupassen und dadurch den Stromverbrauch zu senken. Die verwendeten Sensorknoten vom Typ TelosB erfassen ihre Messwerte einmal pro Sekunde und leiten diese alle 10 Sekunden an eine Datensenke weiter.

Diese drei Beispiele zeigen, wie durch das periodische Weiterleiten der vom Sensornetz gesammelten Messwerte an eine Datensenke ein Gebäudeüberwachungssystem realisiert werden kann.

7.2. Demonstrator

Dieser Abschnitt beschreibt den Demonstrator, der zur Realisierung des Anwendungsszenarios umgesetzt wurde. Der Demonstrator besteht hierbei aus insgesamt zwei Teilen: Der Sensornetzteil der Anwendung, der in Abschnitt 7.2.1 beschrieben ist und der für die Datenerfassung und Weiterleitung der Nachrichten an die Datensenke sowie die Speicherung in einer Datenbank zuständig ist. Anschließend wird in Abschnitt 7.2.2 die Benutzungsschnittstelle des Demonstrators beschrieben, die für den Zugriff auf die Daten und deren Visualisierung zuständig ist.

7.2.1. Sensornetzwerk

Als Testumgebung dient das WISEBED-Testbed am Institut für Telematik (vgl. Abschnitt 2.1.6). Es wurde hierbei dieselbe Implementierung des DPS-Protokolls verwendet, die bereits in den Evaluationen aus Kapitel 6 verwendet wurde, so dass sich die in Abschnitt 5.6 beschriebene Aufgabenteilung zwischen Client und Server ergibt. Hierbei übernahmen die Sensorknoten vom Typ JN5139 wieder die Rolle des DPS-Clients, während die Sensorknoten vom Typ JN5148 wieder die Rolle der DPS-Server übernahmen.

Die Sensorknoten sind mit Temperatur-, Luftfeuchtigkeits- und Helligkeitssensoren sowie Bewegungsmeldern ausgestattet, wobei jeder Sensorknoten jeweils nur über eine Untermenge dieser Sensoren verfügt. Wie in den verwandten Arbeiten zu Beginn dieses Kapitels vorgestellt wurde, können diese Sensoren zur bedarfsgerechten Steuerung der Heizungs- und Lüftungsanlage sowie der Beleuchtung verwendet werden, indem z.B. die Bewegungsmelder zur Anwesenheitserkennung von Personen genutzt werden.

Zusätzlich zu diesen Messwerten sammeln die Sensorknoten Zustandsinformationen über sich selbst und das umgebende Netzwerk. Zu diesen Statusinformationen gehört unter anderem der aktuell belegte und verfügbare Arbeitsspeicher, die eigene MAC-Adresse sowie ein Nachrichtenzähler. Darüber hinaus speichert jeder Sensorknoten die in jedem Intervall von ihm versendete und empfangene Datenmenge, die Anzahl der benachbarten Sensorknoten, sowie die Anzahl von Sensorknoten, zu denen er aktuell eine DPS-Verbindung aufgebaut hat.

Die Gesamtheit dieser Messwerte und Statusinformationen wird in einer Report-Nachricht gesammelt und regelmäßig an die Datensenke mittels UDP weitergeleitet. Die Datensenke leitet sämtliche empfangenen Report-Nachrichten an eine Server-Komponente weiter, welche die empfangenen Daten in einer SQL-Datenbank speichert. Neben den in der Report-Nachricht enthaltenen Werten wird in der Datenbank zusätzlich noch der Empfangszeitstempel (Uhrzeit und Datum) der Nachricht an der Datensenke gespeichert.

7.2.2. Benutzungsschnittstelle

Um auf die Messwerte des Sensornetzes zugreifen zu können, wurde eine Webanwendung auf Basis der Google Visualization-API[1] realisiert. Hierfür wurde zunächst ein Adapter implementiert, der die in der SQL-Datenbank gespeicherten Messwerte zur Verfügung stellen kann. Dieser Adapter kann eine Untermenge der SQL-Abfragesprache auf die in der Datenbank gespeicherten Messwerte anwenden und die resultierenden Messwerte im JSON-Format (JavaScript-Object-Notation) über eine REST-Schnittstelle zur Verfügung stellen. Das hierbei verwendete JSON-Format entspricht der von der Google-Visualization-API benötigten Darstellung.

Die Datasource unterstützt die Google-Visualization-API-Query-Language, die eine Untermenge der SQL-Datenbanksprache darstellt. Hierbei werden die Schlüsselwörter SELECT und WHERE sowie einige Gruppierungs- und Filteroptionen unterstützt (GROUP BY, ORDER BY, LIMIT, OFFSET), während verändernde Befehle wie UPDATE, DELETE oder INSERT nicht unterstützt werden. Es besteht folglich lesender, aber kein schreibender Zugriff auf die Datenbank. Auf diese Weise sind z.B. Anfragen der Art „SELECT x WHERE y“ möglich, während die Auswahl der Tabelle (der FROM Parameter der SQL Abfrage) nicht möglich ist. Ein Beispiel für eine solche Anfrage ist in Abbildung 7.1 dargestellt, die alle am 6. November 2013 gesammelten Temperaturwerte des Sensorknoten 0x213C zurückliefert.

```
1 SELECT time, temperature WHERE ((node = '0x213C')
2           AND (time >= '2013-11-06␣00:00:00')
3           AND (time <= '2013-11-06␣23:59:59'))
```

Abbildung 7.1.: Beispiel einer Abfrage zum Abrufen aller Temperaturwerte des Sensorknoten 0x213C am 6. November 2013

Die Webanwendung bietet dem Benutzer über ein Webinterface eine Liste aller verfügbaren Sensoren, von denen der Benutzer eine Untermenge zur Anzeige auswählen kann. Der Benutzer kann zusätzlich ein Datum auswählen und bekommt anschließend sämtliche von den ausgewählten Sensorknoten gesammelten Messwerte des gewählten Tages in Form eines Graphen dargestellt. Der Graph zeigt hierbei den gewählten Messwert eines Sensorknotens in Abhängigkeit von der Uhrzeit in einem zoom- und scrollbaren Graphen an. Auf diese Weise kann der Benutzer den täglichen Verlauf der Messwerte zwischen den einzelnen Sensorknoten vergleichen. Er kann jedoch auch die Messwerte jedes einzelnen Sensorknotens genauer analysieren, indem er an einen beliebigen Zeitpunkt hineinzoomt. Dieses Webinterface ist jedoch nur eine der möglichen Verarbeitungsarten, da die von der Google-Visualization-API zur Verfügung gestellte Schnittstelle auch eigenständig genutzt werden kann und durch die Query-Language auch komplexe Anfragen ermöglicht.

Ein Beispiel für die Visualisierung der Messwerte über das Webinterface ist in Ab-

[1] https://developers.google.com/chart/interactive/docs/reference

Abbildung 7.2.: Browser-Darstellung der Anzahl DPS-Verbindungen von Knoten 0x2001 (Client, oben) und 0x2130 (Server, unten) im Verlauf eines Tages

bildung 7.2 dargestellt, welches den Verlauf der Anzahl der DPS-Verbindungen von zwei Sensorknoten zeigt. Knoten 0x2001 (oberes Diagramm in Abbildung 7.2) ist ein DPS-Client und verfügt über die gesamte dargestellte Zeitspanne von 24 Stunden über genau eine DPS-Verbindung. Knoten 0x2130 (unteres Diagramm in Abbildung 7.2) hingegen ist ein DPS-Server, dessen Anzahl an DPS-Verbindungen im Verlauf des Tages zwischen null und drei Verbindungen schwankt.

7.3. Evaluation

Dieser Abschnitt stellt die Ergebnisse der Evaluation des DPS-Protokolls in dem durch das Anwendungsszenario erzeugten Multi-Hop-Szenario vor. Zu Beginn wird zunächst in Abschnitt 7.3.1 die vom DPS-Protokoll in Zusammenhang mit dem Routing-Protokoll erzeugte Netzwerktopologie vorgestellt. Anschließend

wird in Abschnitt 7.3.2 die Dauer des Verbindungsaufbaus des DPS-Protokolls analysiert, gefolgt von der Dauer der einzelnen DPS-Verbindungen. Als nächstes wird in Abschnitt 7.3.3 die Paketankunftsrate sowie die Menge der Duplikate anhand der Nachrichten untersucht, die von den Sensorknoten zur Datensenke weitergeleitet werden. Den Abschluss der Evaluation bildet eine Untersuchung der Round-Trip-Time, die mithilfe von Echo-Request- und Echo-Response-Nachrichten in Abschnitt 7.3.4 untersucht wird.

7.3.1. Topologie

Für die Evaluation des DPS-Protokolls in dem gewählten Anwendungsszenario wurden insgesamt 43 Sensorknoten aus dem WISEBED-Testbed eingesetzt, wovon 16 Knoten vom Typ JN5148 als Server und 29 Knoten vom Typ JN5139 als Client für das DPS-Protokoll verwendet wurden. Die Sensorknoten befinden sich am Institut für Telematik der Universität zu Lübeck und sind über insgesamt 16 Büroräume verteilt. Als Routing-Protokoll wurde das in Abschnitt 5.7 vorgestellte zentralisierte Routing-Protokoll verwendet. Eine Darstellung der gewählten Netzwerktopologie, die für den Versand der Datenpakete zur Senke (Knoten 0x2130 oben rechts) verwendet wurde, befindet sich in Abbildung 7.3. In dieser Abbildung ist die Aufteilung der Sensorknoten auf die Büroräume des Instituts dargestellt, wobei die DPS-Server als rote Rechtecke und Clients als blaue Dreiecke dargestellt sind. Die Linien geben die gewählten Verbindungen der Knoten in Richtung der Senke an, wobei eine gestrichelte Linie eine DPS-Verbindung zwischen einem Server und einem Client darstellt, während eine durchgezogene Linie eine IP-Verbindung zwischen zwei Servern darstellt, die mittels des Routing-Protokolls gewählt wurde. Die in dieser Abbildung dargestellten Links stellen hierbei die durchschnittlich gewählte Topologie dar, d.h. für jeden der insgesamt 43 Sensorknoten wurde für jedes gesendete UDP-Paket über die gesamte Dauer der Experimente die Adresse des nächsten in Richtung der Senke liegenden Knotens gespeichert. Im Anschluss an die Experimente wurde dann ausgewertet, welches der am häufigsten ausgewählte Knoten war, um die durchschnittliche Topologie zu ermitteln. Im Falle eines Clients ist dies derjenige

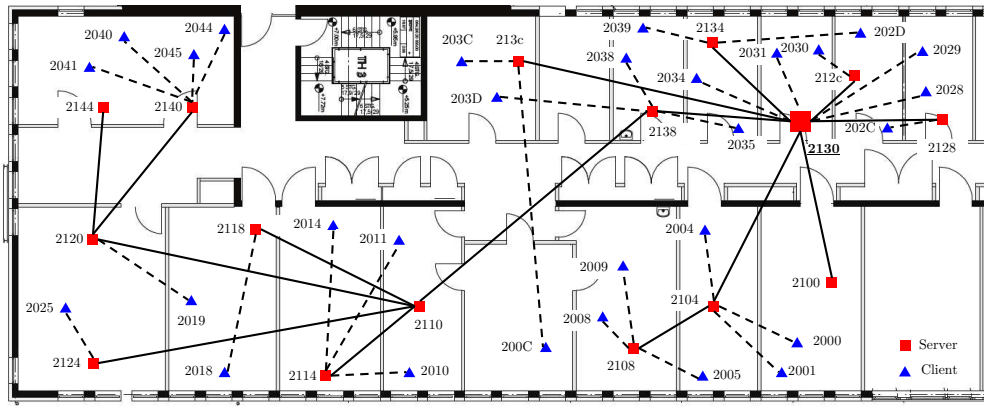


Abbildung 7.3.: Visualisierung der durchschnittlich gewählten Topologie

Server, zu dem am längsten eine DPS-Verbindung aufgebaut wurde. In der resultierenden Topologie besitzen alle Knoten eine Entfernung von maximal fünf Hops zur Senke (durchschnittlich 2,4 Hops). Die in Abbildung 7.3 dargestellte Topologie stellt somit die häufigsten Verbindungen der Knoten im Netzwerk dar, die in der dargestellten Kombination nicht notwendigerweise aufgetreten sind, die jedoch einzeln betrachtet am häufigsten aufgetreten sind.

7.3.2. DPS-Verbindungen

Die im vorangegangenen Abschnitt beschriebene Topologie wurde einerseits durch das zentralisierte Routing-Protokoll und andererseits durch das DPS-Protokoll zwischen den Sensorknoten bestimmt. Während das zentralisierte Routing-Protokoll durch ein beliebiges anderes Protokoll, wie z.B. DYMO oder RPL, ausgetauscht werden kann, ist der Einfluss des DPS-Protokolls auf die Wahl der Verbindungen zwischen den Knoten im Rahmen dieser Evaluation von Interesse. Ziel dieses Abschnittes ist es deshalb, diesen Einfluss genauer zu untersuchen. Aus diesem Grund werden in diesem Abschnitt zunächst die Dauer des hierfür benötigten Verbindungsaufbaus und anschließend die Dauer der hergestellten Verbindungen untersucht.

Dauer des Verbindungsaufbaus

Bevor der Client und der Server Daten über eine DPS-Verbindung austauschen können, muss die Verbindung zunächst aufgebaut werden. Hierfür wird, wie in Abschnitt 5.1 beschrieben, zunächst das Discovery- und Advertisement-Protokoll und anschließend der Drei-Wege-Handschlag durchlaufen. Die für den Demonstrator verwendete Version des DPS-Protokolls ist hierbei so konfiguriert worden, dass der Client mindestens $K_D = 3$ Discovery-Nachrichten im Abstand von $T_D = 30\,\text{ms}$ versendet (vgl. Abschnitt 5.1.1), bevor er mit dem Drei-Wege-Handschlag zu dem für ihn optimalen Server beginnt. Als Metrik für die Wahl des Servers wird im Demonstrator der LQI verwendet, da diese Metrik laut der Ergebnisse in der Arbeit von [169], welche die Ergebnisse mehrerer anderer Arbeiten zu diesem Thema zusammenfasst, eine stärkere Korrelation zur Paketankunftsrate und -fehlerrate eines Links aufweist als die RSSI-Metrik.

Anschließend beginnt der Server mit dem Drei-Wege-Handschlag. Um die Zeit zu messen, die für den Verbindungsaufbau benötigt wird, misst jeder Client die Zeit $T_{CONNECT}$ zwischen dem Beginn des Discovery-Prozesses und dem erfolgreichen Verbindungsaufbau (Versand der Finish-Nachricht als Antwort auf die Allow-Nachricht des Servers) und gibt diese beim Aufbau einer Verbindung über das WISEBED-Testbed an ein Auswertungsscript weiter. Aus den auf diese Weise gesammelten Daten wurde ein Histogramm erstellt, das in Abbildung 7.4 dargestellt ist. Durch die Wahl der Parameter K_D und T_D beträgt die untere Schranke für die Dauer des Verbindungsaufbaus $K_D * T_D = 90\,\text{ms}$ plus der Zeit, die für den Versand der Nachrichten des Drei-Wege-Handschlags und deren Verarbeitung notwendig ist. Wie aus Abbildung 7.4 hervorgeht, beträgt die mini-

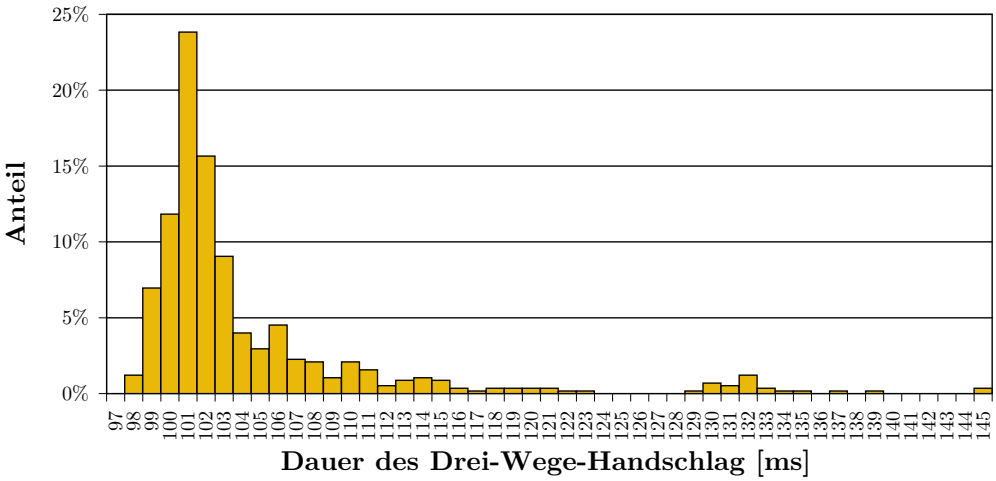


Abbildung 7.4.: Histogramm der Dauer des Verbindungsaufbaus

male gemessene Dauer 98 ms, während der größte Anteil 101 ms benötigte, was die Schlussfolgerung zulässt, dass das Versenden und Verarbeiten der insgesamt drei Nachrichten des Drei-Wege-Handschlag minimal 8 ms und meistens 11 ms dauert. Die Verteilung des Histogramms wird durch den CCA-Mechanismus (Clear Channel Assessment) der 802.15.4-Sicherungsschicht hervorgerufen, der pro Nachricht mindestens einmal und maximal viermal durchlaufen wird, wobei sich die Länge exponentiell erhöht (0-2,5 ms, 0-5 ms, 0-10 ms 0-20 ms). Hierdurch ergibt sich der Abfall des Histogramms links und rechts des Maximums.

Das zweite, deutlich kleinere, Maximum bei 132 ms hingegen ist das Ergebnis des Discovery-Prozesses. Dieser wird minimal $K_R = 3$-mal durchlaufen und anschließend wiederholt, falls bei den ersten drei Durchläufen kein einziges Advertisement von einem Server empfangen wurde. In diesem Fall verlängert sich die Dauer des Verbindungsaufbaus um weitere $T_D = 30$ ms, was jedoch nur in 5,4 % aller Fälle passiert.

Dauer einer Verbindung

Nachdem eine DPS-Verbindung aufgebaut wurde, wird diese mittels des in Abschnitt 4.1.5 beschriebenen Mechanismus durch das Versenden von Heartbeat-Nachrichten überwacht, um auf den Abbruch einer Verbindung reagieren zu können. Für den vorliegenden Anwendungsfall wurde das DPS-Protokoll so konfiguriert, dass ein Verbindungsabbruch angenommen wird, sobald länger als T_C=4000 ms keine Nachricht mehr empfangen wurde, wobei beide Kommunikationspartner regelmäßig alle T_H=1000 ms Heartbeat-Nachrichten austauschen. Es müssen als mindestens vier Heartbeat-Nachrichten eines Knotens hintereinander verloren gehen, um einen Verbindungsabbruch zu erkennen. Die Wahl der Parameter wurde in Abschnitt 5.1.6 genauer beschrieben.

Um die Verbindungsdauer im Anwendungsszenario zu untersuchen, melden alle Clients den Zeitpunkt eines Verbindungsabbruchs über das WISEBED-

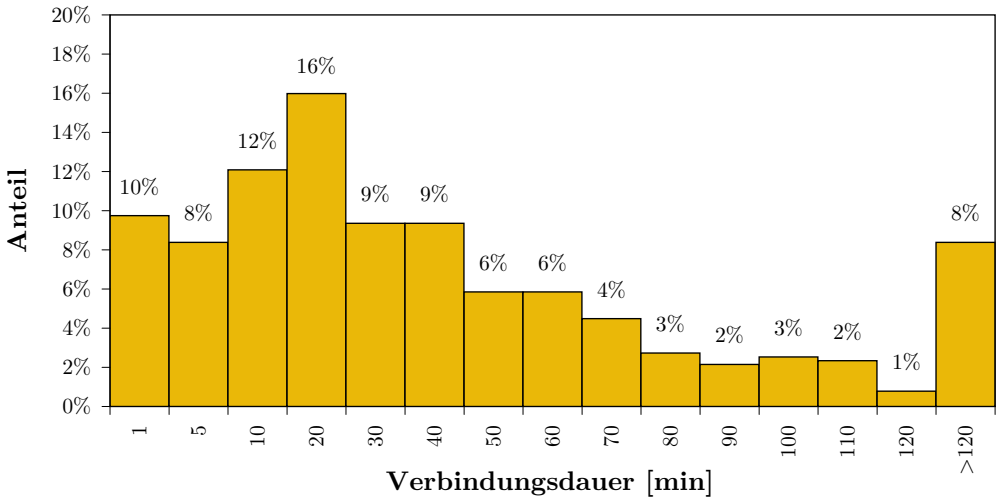


Abbildung 7.5.: Histogramm der Dauer aller DPS-Verbindungen im Netzwerk

Testbed. Im Zusammenhang mit dem Zeitpunkt des von diesem Sensorknoten zuvor gemeldeten Verbindungsaufbaus kann das Auswertungsskript die Dauer der DPS-Verbindung berechnen. Die Ergebnisse sind in Abbildung 7.5 dargestellt, das ein Histogramm der ermittelten Verbindungsdauern während des Experiments zeigt. Aus dem Diagramm geht hervor, dass der Großteil aller Verbindungen (17 %) eine Dauer zwischen 10 min und 20 min aufweist, während 8 % aller Verbindungen eine Dauer von mehr als 120 min aufweisen. Dieses Histogramm vermittelt einen Überblick über die generelle Verbindungsdauer während der Experimente, die durch die Eigenschaften der drahtlosen Verbindung zwischen den Sensorknoten bestimmt wird. Dies wird unter anderem aus der großen Streuung der Verbindungsdauer von weniger als einer Minute bis zu über zwei Stunden deutlich, wobei die maximale Verbindungsdauer während der Experimente 8 Stunden und 30 Minuten betrug.

Eine Aufschlüsselung dieser Daten findet sich in Abbildung 7.6, welche die durchschnittliche Anzahl an neuen Verbindungen pro Stunde für jeden Sensorknoten angibt, getrennt nach Servern (blau) und Clients (rot). Hierbei ist zu erkennen, dass Server eine durchschnittlich höhere Anzahl an neuen Verbindungen pro Stunde aufweisen (2,0) als Clients (1,1). Dies ist dadurch zu erklären, dass jeder Server eine Verbindung zu mehreren Clients aufbauen kann, während Clients immer nur zu genau einem Server eine DPS-Verbindung aufbauen können. Aufgrund des Zahlenverhältnisses von etwa 1:2 bei 16 Servern zu 29 Clients, muss jeder Server eine Verbindung zu durchschnittlich 1,8 Clients besitzen. Darüber hinaus fällt auch hier eine starke Streuung zwischen den einzelnen Sensorknoten auf, wobei diese bei den Clients zwischen 0,6 und 2,0 sowie bei den Servern zwischen 1,2 und 3,1 neuen Verbindungen pro Stunde schwankt. An der großen Schwankung der Clients lässt sich im Zusammenhang mit den Daten aus Abbildung 7.5 darauf schließen, dass die Ursache für die Verbindungsabbrüche die Eigenschaften der drahtlosen Kommunikation zwischen den Clients und ihrem Server sind: Einige Sensorknoten sind an einer günstigeren Stelle im Netzwerk positioniert, so dass sie durchschnittlich eine fast drei- bis viermal länger dau-

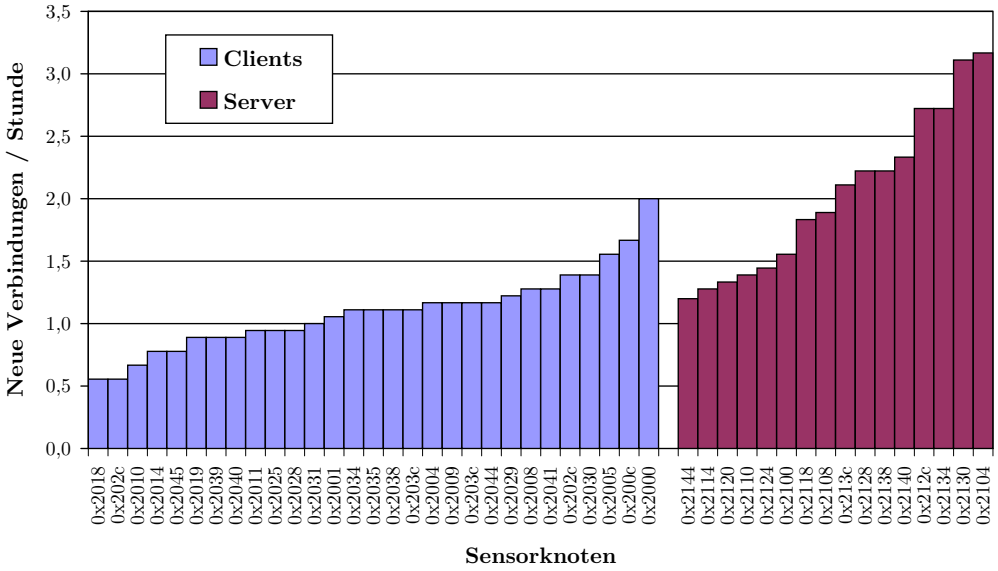


Abbildung 7.6.: Anzahl neuer Verbindungen pro Stunde für alle Knoten im Netzwerk, getrennt nach Clients (links, blau) und Servern (rechts, rot)

ernde Verbindung zu ihrem Server verwenden. Die Dauer einer DPS-Verbindung hängt folglich von der Position eines Sensorknotens ab - ist die Paketverlustrate niedrig und die Anzahl von Kollisionen auf dem Funkmedium gering, so ist die Verbindungsdauer länger als bei einer hohen Paketverlustrate oder einer hohen Anzahl von Kollisionen.

7.3.3. Paketankunftsrate

Nachdem die DPS-Verbindung aufgebaut wurde, können sowohl Clients als auch Server die von ihnen gesammelten Messwerte an die Datensenke weiterleiten. Da zum Datenversand im Rahmen des Demonstrators das UDP-Protokoll verwendet wird, kommen hierbei keine Ende-zu-Ende-Mechanismen zur Paketverlusterkennung und -kompensation auf der Transportschicht zum Einsatz. Da jedoch der iSense IP-Stack den Bestätigungsmechanismus der 802.15.4-Sicherungsschicht verwendet, kann der Nachrichtenverlust auf der Sicherungsschicht erkannt und korrigiert werden. Da der Benutzer im Anwendungsfall der Gebäudeüberwachung an den für ihn sichtbaren Daten interessiert ist, wird in diesem Abschnitt die Paketankunftsrate (PRR) an der Datensenke untersucht. Hierfür werden die in allen Report-Nachrichten enthaltenen Paketzähler verwendet: Zu Beginn des jeweiligen Experiments initialisieren alle Sensorknoten ihren Paketzähler auf null und erhöhen diesen bei jeder gesendeten Report-Nachricht um eins. Auf diese Weise kann an der Datensenke durch das Zählen der fehlenden Pakete in einem gewählten Zählerbereich die PRR jedes einzelnen Sensorknotens zu jedem Zeitpunkt berechnet werden.

Aus der auf diese Weise berechneten PRR soll nun zunächst die allgemeine

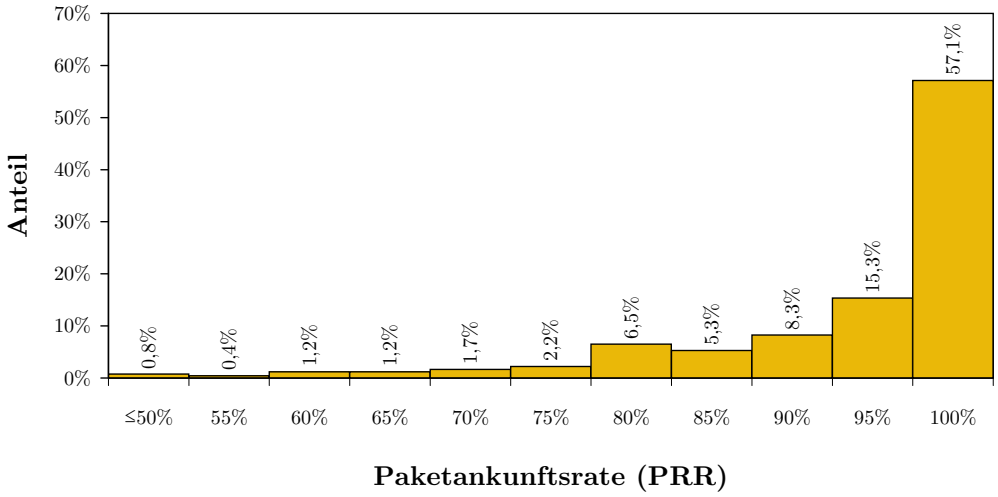


Abbildung 7.7.: Histogramm der Paketankunftsrate im gesamten Netzwerk

PRR des gesamten Sensornetzes ermittelt werden. Diese ist in Abbildung 7.7 in der Form eines Histogramms dargestellt und zeigt die Verteilung der PRR im Bereich von 50 % bis 100 % in Schritten von 5 % über die Gesamtlaufzeit der durchgeführten Experimente. Die PRR wurde hierbei für jeden Sensorknoten individuell in einem 25 Pakete langen Fenster berechnet. Die Ergebnisse in Abbildung 7.7 zeigen, dass die PRR in 57,1 % aller Fälle zwischen 95 % und 100 % liegt und in 80,7 % aller Fälle oberhalb von 90 % liegt (8,3 % + 15,3 % + 57,1 %), während nur in 0,8 % aller Fälle eine PRR von höchstens 50 % erreicht wird. Dieser Überblick über die allgemeine PRR im Netzwerk zeigt, dass das DPS-Protokoll im Zusammenspiel mit dem IP-Protokoll eine hohe PRR erzielt.

Neben der allgemeinen ist die individuelle PRR der einzelnen Knoten von Interesse, so dass einerseits eine Unterscheidung zwischen den über das DPS-Protokoll und den über das Internet-Protokoll verschickten Daten möglich ist und andererseits der Einfluss der Position der einzelnen Knoten berücksichtigt werden kann. Aus diesem Grund wurden die für Abbildung 7.7 gesammelten Daten zusätzlich getrennt für die einzelnen Sensorknoten untersucht. Das Ergebnis ist in Abbildung 7.8 dargestellt, das die PRR jedes einzelnen Sensorknotens über die gesamte Dauer der Experimente zeigt. Neben der MAC-Adresse jedes Sensorknotens ist auf der X-Achse die durchschnittliche Entfernung des entsprechenden Sensorknotens zur Datensenke in Hops angegeben, die von minimal 1,0 Hops bis maximal 4,6 Hops reicht. Zur Darstellung in der Abbildung wurden die Knoten in aufsteigender Reihenfolge sortiert, wobei links einer geringen Entfernung und rechts einer großen Entfernung zur Datensenke entspricht. Zusätzlich zu der PRR wurde auch die Menge der doppelt an der Datensenke empfangenen Nachrichten (Duplikate in Abbildung 7.7) für jeden einzelnen Sensorknoten ermittelt.

Die Ergebnisse aus Abbildung 7.8 zeigen, dass von den 44 Sensorknoten mehr als 70 % der Knoten (31) eine durchschnittliche PRR von mindestens 95 % aufweisen und nur ein einziger Sensorknoten eine PRR von weniger als 90 %

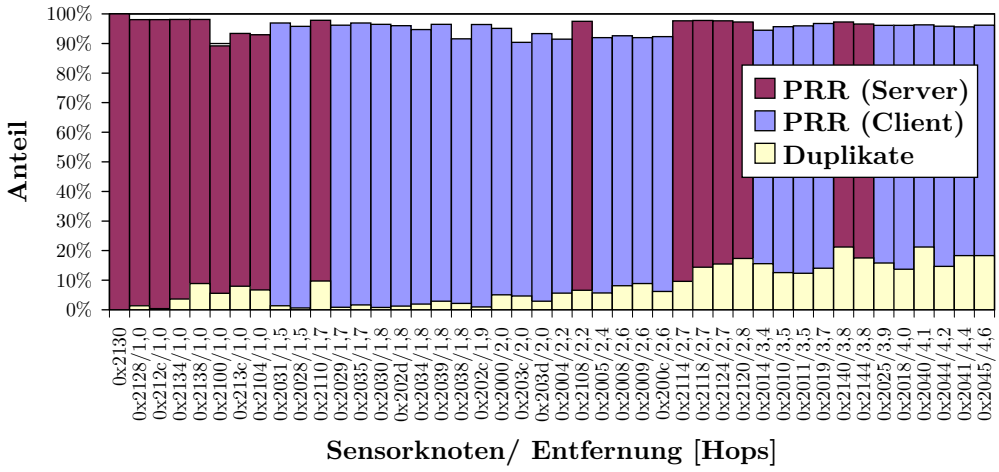


Abbildung 7.8.: Paketankunftsrate und Duplikate pro Knoten in Abhängigkeit von der Entfernung zur Senke

aufweist (Knoten 0x2100 mit 89,2 %). Insgesamt sinkt die PRR mit steigender Entfernung zur Datensenke leicht, während sich die Menge der Duplikate erhöht. Dieser Effekt ist durch die fehlende Verlusterkennung des UDP-Protokolls zu erklären, das außerdem keinen Mechanismus zur Duplikaterkennung bietet, da UDP keine Nachrichtenzähler verwendet (siehe unten). Der direkte Vergleich der PRR der Server- und Client-Knoten im Netzwerk zeigt, dass die Server eine durchschnittlich höhere PRR ausweisen, als die Clients. Die Server im Netzwerk verfügen darüber hinaus über eine durchschnittlich geringere Entfernung zur Datensenke, da ein Client eine Report-Nachricht immer zunächst an einen Server schicken muss, bevor diese an die Datensenke weitergeleitet werden kann. Gleichzeitig ist zu erkennen, dass die PRR mit steigender Entfernung nur geringfügig abnimmt, während es sich bei dem Knoten mit der niedrigsten PRR um einen Server mit einer Entfernung von einem Hop zur Datensenke handelt (0x2100). Dies lässt den Schluss zu, dass die PRR von der Position des Sensorknotens in der Netzwerktopologie abhängig ist und keine negativen Auswirkungen des DPS-Protokolls messbar sind. Dies zeigen vor allem die Ergebnisse der Client-Knoten mit einer Entfernung von mehr als drei Hops zur Datensenke, deren PRR höher ist als die der Client-Knoten mit einer Entfernung von zwei Hops.

Die in Abbildung 7.8 dargestellten Daten zeigen darüber hinaus, dass die Menge der Duplikate bei sinkender PRR steigt. Dieser Effekt wird durch den Bestätigungsmechanismus der Sicherungsschicht hervorgerufen, der eine Nachricht mehrere Male sendet, falls die zugehörige Bestätigung nicht empfangen wurde. Geht jedoch nicht die Nachricht selbst, sondern die zugehörige Bestätigung verloren, so wird die Nachricht nicht nur mehrmals gesendet, sondern auch mehrmals empfangen. Da UDP keinen Mechanismus zur Filterung von Duplikaten verwendet, werden diese Duplikate anschließend bis zur Datensenke weitergeleitet. Bei kurzen Distanzen zur Datensenke (<2 Hops) fällt hierbei auf, dass die Menge der Duplikate bei den Clients deutlich geringer ausfällt als

bei den Servern, obwohl ihre PRR schlechter ist. Ein Beispiel bilden der Client 0x2038 und der Server 0x2110: Der Client hat eine Entfernung von 1,8 Hops zur Datensenke und eine PRR von 91,6 %, während der Server eine Entfernung von 1,7 Hops und eine PRR von 97,9 % aufweist. Obwohl dieser Server eine um 6,3 % höhere PRR aufweist, besitzt er eine um 7,7 Prozentpunkte höhere Menge an Duplikaten (9,8 % gegenüber 2,1 %). Dies ist eine positive Auswirkung des DPS-Protokolls, das einen Mechanismus zur Filterung von Duplikaten aufweist. Die Auswirkungen dieses Mechanismus sind jedoch nur für kurze Distanzen messbar, da bei größeren Distanzen das Verhältnis von DPS-Hops (maximal 1) zu IP-Hops (maximal 3.6) schlechter wird.

7.3.4. Round-Trip-Time

Zusätzlich zu der Evaluation der Single-Hop-RTT in Abschnitt 6.5 soll in diesem Abschnitt untersucht werden, wie stark die Auswirkungen des DPS-Protokolls in einem Multi-Hop-Szenario sind. Analog zu den Single-Hop-Experimenten wurde auch für diese Experimente das ICMP-Protokoll verwendet, mit dem ausgehend von einem mit der Datensenke verbundenen DPS-Client-Knoten ICMP-Echo-Request-Pakete an einen ausgewählten Sensorknoten im Netzwerk gesendet wurden, der mit einer Echo-Response-Nachricht antwortet. Hierbei wurde die Zeit gemessen, die zwischen dem Senden des Echo-Request und dem Empfang der Echo-Response-Nachricht vergeht. Dieser Vorgang wurde insgesamt 100-mal für jeden der 43 Sensorknoten im Netzwerk wiederholt, so dass insgesamt 4300 ICMP-Echo-Request-Nachrichten verschickt wurden. Jedes dieser ICMP-Pakete verfügt über eine Payload-Länge von 48 Byte, was dem maximalen Payload in einem Fragment des DPS-Protokolls entspricht (vgl. Abschnitt 6.5). Als Quelle für die ICMP-Pakete wurde ein DPS-Client ausgewählt, der mit der Datensenke verbunden ist, damit die Route entlang der ICMP-Pakete jeweils mindestens einen DPS-Hop enthält. Falls es sich bei dem Ziel des ICMP-Pakets ebenfalls um einen DPS-Client handelt, erhöht sich die Anzahl der DPS-Hops entlang dieser Route auf zwei. Im Rahmen dieser Experimente wurde das zentralisierte Routing-Protokoll der vorherigen Experimente durch ein statisches Routing ersetzt, das die in Abbildung 7.3 dargestellte durchschnittliche Topologie umsetzt. Auf diese Weise wurde sichergestellt, dass die ICMP-Pakete immer dieselbe Route wählen.

Die Ergebnisse dieser Experimente sind in Abbildung 7.9 zusammengefasst, welche die RTT in Abhängigkeit von der Entfernung in Hops zeigt. Hierbei zeigt der Balken jeweils den Median der gemessenen RTT des entsprechenden Sensorknotens an, während die Antennen (Whisker) den oberen und unteren Quartilsabstand anzeigen (zwischen dem oberen und dem unteren Quartil befinden sich 50 % der Messwerte). Die Sensorknoten wurden nach ihrer Entfernung zur Datensenke sortiert, wobei die Sensorknoten links eine geringere Entfernung besitzen als die Knoten auf der rechten Seite (bis zu sechs Hops). Anschließend wurden die Sensorknoten in jedem Hop nach dem Median ihrer RTT sortiert. Um die Hops besser voneinander abzuheben, wurde eine Pause zwischen den einzelnen Hops eingefügt. Zur besseren Unterscheidbarkeit wurden zusätzlich

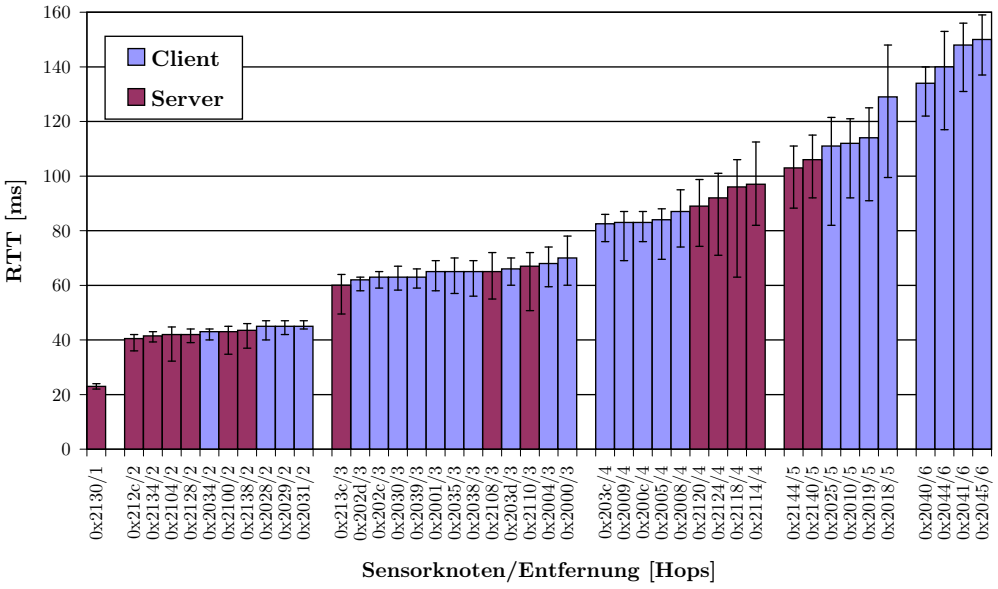


Abbildung 7.9.: Multi-Hop-RTT entlang der Topologie aus Abbildung 7.3

die Balken der Clients blau und die Server rot eingefärbt.

Die Ergebnisse in Abbildung 7.9 zeigen, dass die RTT mit steigender Entfernung zunimmt, wobei sich die RTT für jeden Hop um etwa 20 ms erhöht. Zusätzlich erhöht sich die RTT innerhalb jedes Hops in Abhängigkeit von der Position des Sensorknoten. Diese zusätzliche Erhöhung beträgt auf dem zweiten Hop bis zu 5 ms, auf dem dritten Hop bis zu 10 ms, auf dem vierten Hop bis zu 20 ms und auf den weiteren Hops bis zu 30 ms. Außerdem ist zu erkennen, dass sich der Abstand zwischen dem oberen und unteren Quartil mit steigender Entfernung ebenfalls erhöht. Der Grund für die Erhöhung der RTT mit steigender Distanz liegt in der größeren Anzahl an Hops, die von einem Paket zurückgelegt werden muss. Mit jedem zusätzlichen Hop erhöht sich zudem der Einfluss des CCA-Mechanismus von CSMA auf die Übertragungszeit, da dieser auf jedem Hop erneut durchlaufen werden muss.

Ein Vergleich der Ergebnisse zwischen den Clients und Servern zeigt, dass nur auf dem zweiten Hop ein geringer Unterschied in der RTT messbar ist. Während in den Single-Hop-Experimenten unter Laborbedingungen noch ein unterschied von durchschnittlich 9 ms zwischen dem DPS-Protokoll und dem nativen IPv6-Protokoll festgestellt werden konnte, beträgt dieser Unterschied in diesen Experimenten auf dem zweiten Hop maximal 4 ms, wobei einige der DPS-Clients sogar eine geringere oder gleich große RTT aufweisen wie die Server. Dieser Unterschied zu den Experimenten unter Laborbedingungen wird ab dem dritten Hop noch deutlicher und kehrt sich im vierten Hop in das Gegenteil um: Die Daten des Vierten Hops zeigen, dass das DPS-Protokoll sogar einen positiven Einfluss auf die RTT hat, da in diesem Hop sämtliche DPS-Clients eine niedrigere RTT als die DPS-Server aufweisen (mindestens 2 ms und bis zu 14 ms). Diese Ergebnisse zeigen, dass die Auswirkungen des DPS-Protokolls nur unter Laborbedingungen messbar sind und in einem realen Anwendungsszenario

von anderen Effekten überlagert werden. Hierzu gehört neben den Eigenschaften des Funkmediums der Einfluss des Hintergrundverkehrs, da jeder Sensorknoten weiterhin die Report-Nachrichten alle 30 s an die Datensenke weiterleitet.

8. Zusammenfassung und Ausblick

Drahtlose Sensornetze haben sich in den letzten Jahren vor allem durch die Entwicklung der 6LoWPAN-Anpassungsschicht und der damit ermöglichten Integration von IPv6 zu einem wesentlichen Bestandteil des Internets der Dinge entwickelt. Durch die Abkehr von den zuvor eingesetzten proprietären Kommunikationsprotokollen wurde eine Grundlage geschaffen, um die Interoperabilität von Sensornetzen unterschiedlicher Hersteller zu ermöglichen. In dieser Arbeit konnte jedoch gezeigt werden, dass die Verwendung von IPv6 auf den stärker ressourcenbeschränkten Plattformen nicht oder nur eingeschränkt möglich ist.

Um den Betrieb von IPv6 trotz der Anforderungen an den Programmspeicher auch auf stärker ressourcenbeschränkten Plattformen zu ermöglichen, wurde in dieser Arbeit ein Konzept entwickelt, das die Kooperation benachbarter Sensorknoten zur Bildung eines verteilten Protokollstapels erlaubt. Das Konzept dieser verteilten Protokollstapel ermöglicht es, dass die einzelnen Sensorknoten in Abhängigkeit des zur Verfügung stehenden Programmspeichers nur einen Teil der benötigten Protokolle selbst implementieren und die fehlenden Protokolle von benachbarten Sensorknoten mittels Remote-Procedure-Calls mitbenutzen. Das DPS-Protokoll sieht hierbei eine Aufgabenteilung zwischen den DPS-Servern und DPS-Clients vor, bei der die Server ihre Implementierung des ausgewählten Protokolls in Form eines Server-Skeletons zur Verfügung stellen, während die Clients diese Implementierung durch die Verwendung eines Client-Stubs über Remote-Procedure-Calls einbinden.

Die Funktionsweise des in dieser Arbeit vorgestellten Konzepts konnte durch die Implementierung des in Kapitel 4 vorgestellten DPS-Protokolls gezeigt werden. Das DPS-Protokoll verfolgt hierbei einen verbindungsorientierten Ansatz, bei dem der Client zunächst die benachbarten Server in seiner unmittelbaren Umgebung erkennt und anschließend eine Verbindung zu dem benötigten Server-Skeleton aufbaut. Das in Kapitel 5 beschriebene Beispiel sieht vor, dass Sensorknoten mit ausreichend Programmspeicher die Rolle der Server einnehmen und den gesamten Protokollstapel implementieren, während die stärker ressourcenbeschränkten Geräte als Clients lediglich einen Teil der Schichten des Protokollstapels implementieren. Durch die gewählte Aufgabenteilung konnten die Anforderungen an den benötigten Programmspeicher für die Clients so weit reduziert werden, dass ein Einsatz von IPv6 möglich ist. Im Gegenzug werden die Anforderungen für die Server nur geringfügig erhöht, ohne die Funktionsweise der Server einzuschränken.

Das DPS-Protokoll benötigt zur Laufzeit lediglich 772 Byte Arbeitsspeicher plus 92 Byte für jede zur Laufzeit aufgebaute Verbindung. Die DPS-Clients benötigen

lediglich eine Verbindung für jedes über die Client-Stubs verwendete Protokoll, so dass der benötigte Arbeitsspeicher in dem vorliegenden Beispiel lediglich 864 Byte beträgt. Die Server hingegen unterstützen mehrere Verbindungen, deren Anzahl lediglich durch den verfügbaren Arbeitsspeicher beschränkt wird. Der beim Versenden und Empfangen von Nachrichten entstehende Overhead ist konstant und unabhängig von der Länge der Nachricht, so dass über das DPS-Protokoll Nachrichten versendet werden können, die größer als die von IPv6 vorgeschriebene MTU von 1500 Byte sind.

Die durch das DPS-Protokoll resultierenden negativen Auswirkungen auf die Latenz und den Datendurchsatz konnten im Rahmen der Single-Hop-Evaluationen in Kapitel 6 genau quantifiziert werden. Die Round-Trip-Time erhöht sich durch den Einsatz des DPS-Protokolls nur um durchschnittlich 9 ms, was in Abhängigkeit von der Nachrichtengröße zwischen 13 % und 33 % der RTT entspricht. Dieser negative Einfluss ist jedoch nur unter Laborbedingungen messbar, wie mittels der Multi-Hop-Evaluationen im Rahmen des Anwendungsbeispiels in Abschnitt 7.3.4 gezeigt werden konnte. Die Multi-Hop-Evaluationen zeigen eine deutlich geringere Verschlechterung der Latenz um maximal 4 ms, die nur für kurze Entfernungen messbar ist. Der Datendurchsatz verringert sich durch das DPS-Protokoll lediglich um 5 %, falls Nachrichten mit einer Länge von mehr als einem Fragment versendet werden sollen.

Durch den Einsatz der Sicherheitsmechanismen erhöht sich die Round-Trip-Time um maximal 10 ms. Der Datendurchsatz hingegen verringert sich nur um weitere 5 % ab einer Paketgröße von mindestens drei Fragmenten. Diese Auswirkungen stellen einen vertretbaren Mehraufwand dar, wenn die Integrität und Authentizität der versenden Nachrichten in der Gegenwart eines Angreifers geschützt werden müssen.

Der Einsatz der Bestätigungen im Rahmen des DPS-Protokolls hat einen deutlich höheren negativen Einfluss auf die Leistungsfähigkeit des Protokolls, weshalb auf den Einsatz der Bestätigungen bei der Datenübertragung verzichtet werden sollte. Bestätigungen sollten nur genau dann eingesetzt werden, falls die RPC-Mechanismen eine Veränderung des Zustands des Protokolls zur Folge haben, wie dies z.B. beim Setzen der IP-Adresse im Rahmen des IPv6-Protokolls der Fall ist.

Es konnte kein nennenswerter negativer Einfluss auf die anderen untersuchten Leistungsparameter, wie z.B. die Paketankunftsrate festgestellt werden. Die Rate der Duplikate konnte durch den im DPS-Protokoll eingebauten Filtermechanismus bei kurzen Entfernungen sogar verbessert werden. Der Vergleich mit den IPv6-Implementierungen des Contiki- und TinyOS-Betriebssystems zeigt, dass das DPS-Protokoll selbst beim kombinierten Einsatz von AES-Prüfsummen und Bestätigungen eine geringere Round-Trip-Time als alle Implementierungen der anderen Betriebssysteme erzielt. Auch der Datendurchsatz ist höher als bei allen verglichenen Implementierungen und sinkt lediglich bei der Verwendung der Bestätigungen des DPS-Protokolls unter den von den anderen Implementierungen erreichten Wert.

Durch den Einsatz der Mechanismen zur Header-Kompression konnte die Anzahl der Fragmente, die zum Versand einer Nachricht einer bestimmten Länge notwendig ist, an das native IPv6-Protokoll angeglichen werden. Die Mechanismen zur Payload-Kompression mittels des S-LZW-Algorithmus hingegen haben zu keiner Verbesserung der Latenz geführt, weshalb von deren Einsatz abgeraten wird.

Die verwendete Aufteilung der Sensorknoten in Server und Clients hat zur Folge, dass Clients immer auf die Anwesenheit von mindestens einem Server in ihrer Funkreichweite angewiesen sind. Dies stellt eine Einschränkung bei der Platzierung der Sensorknoten dar, falls eine kontrollierte Knotenplatzierungsstrategie gewählt werden kann. Ist hingegen nur eine zufällige Verteilung der Sensorknoten im ausgewählten Anwendungsfall möglich, so ergeben sich die in Abschnitt 6.1 beschriebenen Anforderungen an das relative Zahlenverhältnis der Clients und Server im Netzwerk, um die gewünschte Konnektivität zu erreichen. Zusätzlich erhöht sich die Entfernung zwischen dem Client und seinem Kommunikationspartner um einen Hop, falls der Server, dessen Protokoll-Implementierung der Client verwendet, sich nicht in Richtung des Kommunikationspartners befindet. Diese Auswirkung kann im Worst-Case auch höher ausfallen, falls der verwendete Server eine topologisch ungünstige Position im Netzwerk einnimmt.

Zukünftige Arbeiten an dem Konzept der verteilten Protokollstapel sollten sich aus diesem Grund vor allem mit dieser Problematik befassen. Eine Möglichkeit zur Lösung dieses Problems besteht darin, den Server in Abhängigkeit von der Entfernung zum jeweiligen Kommunikationspartner zu wählen. Im Fall des Anwendungsgebiets in Kapitel 7 kann dies eindeutig über die Entfernung des Servers zur Datensenke gelöst werden. Ein Client verwendet in diesem Fall beim Verbindungsaufbau zusätzlich zur Signalstärke die Entfernung des Servers zur Datensenke als Metrik. Existiert in dem Anwendungsfall keine allgemeine Kommunikationsrichtung, z.B. weil die Knoten vor allem untereinander kommunizieren, kann ein Client eine Verbindung zu mehreren Servern aufbauen und in Abhängigkeit vom jeweiligen Zielknoten zwischen den verschiedenen Verbindungen wählen.

Es sollte untersucht werden, ob die negativen Auswirkungen auf die Konnektivität im Netzwerk, die im Rahmen der Simulationen zur zufälligen Knotenplatzierung in Abschnitt 6.1 vorgestellt wurden, durch die Integration von zusätzlichen Strategien zur Nachrichtenweiterleitung ausgeglichen werden können. In der vorliegenden Implementierung ist ein Client auf die Anwesenheit eines Servers in unmittelbarer Funkreichweite angewiesen. Diese Einschränkung könnte durch einen Weiterleitungsmechanismus reduziert werden, bei dem der jeweilige Client auch indirekt über einen benachbarten Client mit dem Server verbunden sein kann, der als eine Art Proxy oder Repeater fungiert. Hierbei sollte untersucht werden, welche Auswirkungen dies auf die Programmgröße und Protokollkomplexität hat und wie stark sich die Einschränkungen des DPS-Protokolls hierdurch verringern lassen. Es sollte jedoch vermieden werden, ein zusätzliches Routing-Protokoll für die Bildung des verteilten Protokollstapels zu implementieren, da hierdurch vermutlich die Vorteile des Konzepts in Bezug auf

die Einsparungen des Programmspeichers aufgehoben werden.

Zusätzlich verfügt die Implementierung des DPS-Protokolls, die im Rahmen dieser Arbeit entstanden ist, vermutlich über weiteres Optimierungspotential. Vor allem der benötigte Programmspeicher kann voraussichtlich durch eine optimierte Implementierung reduziert werden. Neben der Implementierung des DPS-Protokolls auf der Vermittlungsschicht sollten zusätzlich noch weitere Stubs und Skeletons z.B. auf der Transportschicht implementiert werden, um diese mit der Vermittlungsschicht vergleichen zu können.

Darüber hinaus ist eine Erweiterung des Konzepts der verteilten Protokollstapel zu einem verteilten Protokollgraphen möglich. Hierbei kann ein Client im Gegensatz zu dem Beispiel auf der Vermittlungsschicht nicht nur auf ein einzelnes Protokoll zugreifen, sondern zwischen unterschiedlichen Protokollen wählen. Durch die parallele Verwendung von Verbindungen zu verschiedenen Skeletons auf derselben Schicht könnte ein Client so in Abhängigkeit von den jeweiligen Anforderungen zwischen IPv4 und IPv6 auf der Vermittlungsschicht oder UDP und TCP auf der Transportschicht wählen. Dies stellt eine Erweiterung des in Abbildung 3.2 auf Seite 32 vorgestellten Konzepts dar. Zusätzlich kann der Protokollstapel nicht nur auf zwei sondern auf mehrere Sensorknoten verteilt werden, was eine weitere Reduktion der Programmgröße für die einzelnen Sensorknoten ermöglicht. Ein Nachteil liegt darin, dass die negativen Auswirkungen des DPS-Protokolls durch das Hinzufügen der Kommunikation mit den zusätzlichen Sensorknoten steigen.

A. Anhang

Eigene Publikationen

[1] P. Rothenpieler, "Poster Abstract: Distributed Protocol Stacks for Wireless Sensor Networks," in *9th European Conference on Wireless Sensor Networks (EWSN 2012)*, (Trento, Italy), February 2012.

[2] P. Rothenpieler and D. Pfisterer, "Introduction to Distributed Protocol Stacks for Wireless Sensor Networks," tech. rep., 11. Fachgespräche Drahtlose Sensornetze, GI/ITG KuVS, Darmstadt, 2012.

[3] P. Rothenpieler and D. Pfisterer, "Towards Distributed Protocol Stacks for Wireless Sensor Networks," in *Proceedings of the 5th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom 2012)*, (Besançon, France), pp. 418–425, November 2012.

[4] P. Rothenpieler, "Reliability Extensions and Multi-Hop Evaluation of Distributed Protocol Stacks," in *Proceedings of the 6th IEEE International Conference on Cyber, Physical and Social Computing (CPSCom 2013)*, (Beijing, China), pp. 931–938, August 2013.

[5] D. Dudek, C. Haas, A. Kuntz, M. Zitterbart, D. Krüger, P. Rothenpieler, D. Pfisterer, and S. Fischer, "A Wireless Sensor Network For Border Surveillance (Demo)," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, (Berkeley, CA), ACM, November 2009.

[6] P. Rothenpieler, D. Krüger, D. Pfisterer, S. Fischer, D. Dudek, C. Haas, A. Kuntz, and M. Zitterbart, "Poster Abstract: FleGSens - Secure Area Monitoring using Wireless Sensor Networks," in *Proceedings of the 4th Safety and Security Systems in Europe*, pp. 136–139, Micro Materials Center Berlin at Fraunhofer Institute IZM and Fraunhofer ENAS, 2009.

[7] P. Rothenpieler, D. Krüger, D. Pfisterer, S. Fischer, D. Dudek, C. Haas, A. Kuntz, and M. Zitterbart, "FleGSens - Secure Area Monitoring using Wireless Sensor Networks," in *Proceedings of the International Conference on Sensor Networks, Information, and Ubiquitous Computing (ICSNIUC 2009), Singapore*, vol. 56, pp. 81–92, World Academy of Science, Engineering and Technology, 2009.

[8] D. Krüger, C. Haas, P. Rothenpieler, D. Dudek, and D. Pfisterer, "Security in Border Control and Area Monitoring," *it - Information Technology, Themenheft Sicherheit in Sensornetzwerken*, vol. 52, no. 6, pp. 340–344, 2010.

[9] P. Rothenpieler, H. Zwingelberg, D. Carlson, A. Schrader, and S. Fischer, "Datenschutz im AAL service system SmartAssist," in *4 Deutscher AAL-*

Kongress Innovative Assistenzsysteme im Dienste des Menschen - Von der Forschung für den Markt (AAL 2011), (Berlin, Germany), 2011.

[10] P. Rothenpieler, C. Becker, and S. Fischer, "Privacy concerns in a remote monitoring and social networking platform for assisted living," in *Sixth International PrimeLife/IFIP Summer School on Privacy and Identity Management for Emerging Internet Applications throughout a Persons Lifetime*, (Helsingborg, Sweden), Springer-Verlag, 2010.

[11] D. Carlson, P. Rothenpieler, and A. Schrader, "An Open Infrastructure and Platform for AAL Services," in *HCI International 2013, 15th International Conference on Human-Computer Interaction (HCI International)*, (Las Vegas, USA), July 2013.

[12] A. Schrader, D. Carlson, and P. Rothenpieler, "SmartAssist - Wireless Sensor Networks for Unobtrusive Health Monitoring," in *Proceedings of the 5th BMI Workshop on Behaviour Monitoring and Interpretation*, vol. 678, (Karlsruhe, Germany), pp. 84–89, 2010.

[13] A. Schrader, P. Rothenpieler, S. Fischer, and J.-M. Träder, "Ambient Socio-Technical Support for Assisted Autonomous Living," *ERCIM News, Special Issue on Ambient Assisted Living*, Oct. 2011.

[14] C. Becker, P. Rothenpieler, and S. Fischer, "Solutions and challenges for the development of a mobile social network," *International Journal for Infonomics (IJI)*, vol. 3, Dec. 2010.

[15] C. Becker, P. Rothenpieler, and S. Fischer, "Socialcompanion: Connecting virtual and physical social networks." Proceedings of the International Conference on Information Society (i-Society), 2010, June 28 - 30, London, UK, Copyright: IEEE, June 2010.

Verzeichnisse

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[1] K. Ashton, "That 'Internet of Things' Thing," *RFID Journal*, 2009. [Online; accessed 07-Januar-2014] http://www.rfidjournal.com/articles/view?4986.

[2] J. P. Conti, "The internet of things," *Communications Engineer*, vol. 4, no. 6, pp. 20–25, 2006.

[3] B. Warneke, M. Scott, B. Leibowitz, L. Zhou, C. Bellew, J. Chediak, J. Kahn, B. Boser, and K. Pister, "An autonomous 16 mm3 solar-powered node for distributed wireless sensor networks," in *Sensors, 2002. Proceedings of IEEE*, vol. 2, pp. 1510–1515 vol.2, 2002.

[4] International Telecommunication Union, The Internet of Things 2009: Executive Summary, "www.itu.int/osg/spu/publications/internetofthings/." [Online; accessed 10-January-2014].

[5] M. Chui, M. Löffler, and R. Roberts, "The Internet of Things," *McKinsey Quarterly*, 2010. [Online; accessed 13-January-2014] http://www.mckinsey.com/insights/high_tech_telecoms_internet/the_internet_of_things.

[6] M. Chui, M. Löffler, and R. Roberts, "Infocomm Technology Roadmap: Internet of Things," *Infocomm Development Authority of Singapore*, nov 2012. [Online; accessed 13-January-2014] http://www.ida.gov.sg/technologyroadmap.

[7] M. Hempstead, M. J. Lyons, D. Brooks, and G.-Y. Wei, "Survey of hardware systems for wireless sensor networks," *Journal of Low Power Electronics*, vol. 4, no. 1, pp. 11–20, 2008.

[8] S. Deering and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," RFC 2460, Network Working Group, Dec. 1998.

[9] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC 4944, Network Working Group, Sept. 2007.

[10] IEEE 802.15 Working Group, "IEEE 802.15 WPAN Task Group 4 (TG4)." http://www.ieee802.org/15/pub/TG4.html.

[11] Bluetooth Special Interest Group, "Bluetooth Specification." https://www.bluetooth.org/en-us/specification/.

[12] International Electrotechnical Commission, "IEC 62591: WirelessHart."

http://www.iec.ch/etech/2011/etech_0711/store-1.htm.

[13] Z. Pei, Z. Deng, B. Yang, and X. Cheng, "Application-oriented wireless sensor network communication protocols and hardware platforms: A survey," in *IEEE International Conference on Industrial Technology (ICIT 2008)*, pp. 1–6, IEEE, 2008.

[14] M. A. M. Vieira, C. N. Coelho Jr, D. C. da Silva Jr, and J. M. da Mata, "Survey on wireless sensor network devices," in *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA'03. IEEE Conference*, vol. 1, pp. 537–544, IEEE, 2003.

[15] ScatterWeb Internetseite, "http://www.mi.fu-berlin.de/inf/groups/ag-tech/projects/Z_Finished_Projects/ScatterWeb/." [Online; accessed 07-January-2014].

[16] Mica2 Wireless Measurement System, "http://www.eol.ucar.edu/isf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf." [Online; accessed 07-January-2014].

[17] Crossbow TelosB Mote Platform, "http://www.willow.co.uk/TelosB_Datasheet.pdf." [Online; accessed 07-January-2014].

[18] Data Sheet: JN5139-001 and JN5139-Z01, "http://www.jennic.com/files/product_briefs/JN-DS-JN5139-001-1v8.pdf." [Online; accessed 07-January-2014].

[19] Data Sheet: JN5148-001, "http://www.jennic.com/files/product_briefs/JN-DS-JN5148-1v8.pdf." [Online; accessed 07-January-2014].

[20] TU Berlin: Energy Efficient Sensor Networks (EYES), "http://www2.tkn.tu-berlin.de/research/eyes/." [Online; accessed 07-January-2014].

[21] SenseNode - Genetlab Sensor Node v2.0, "http://www.genetlab.com/ENG/product_sensenode.aspx." [Online; accessed 07-January-2014].

[22] Epic: An Open Mote Platform for Application-Driven Design, "http://www.eecs.berkeley.edu/~prabal/projects/epic/." [Online; accessed 07-January-2014].

[23] Tinynode Internetseite, "http://www.tinynode.com/." [Online; accessed 07-January-2014].

[24] Atmel AVR Raven, "http://www.atmel.com/tools/avrraven.aspx." [Online; accessed 07-January-2014].

[25] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks, "http://www.btnode.ethz.ch/." [Online; accessed 07-January-2014].

[26] IRIS-Mote Datenblatt, "http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf." [Online; accessed 07-January-2014].

[27] Libelium Waspmote, "http://www.libelium.com/products/waspmote/

hardware/." [Online; accessed 07-January-2014].

[28] Arduino Internetseite, "http://arduino.cc/." [Online; accessed 07-January-2014].

[29] Sun SPOT Wireless Sensor Node, "http://www.sunspotworld.com/." [Online; accessed 07-January-2014].

[30] Egs: A Cortex M3-based Mote Platform, "www.cs.jhu.edu/~jgko/papers/egs-secon-demo.pdf." [Online; accessed 07-January-2014].

[31] Preon32 Datenblatt, "http://www.virtenio.com/de/preon32.html." [Online; accessed 07-January-2014].

[32] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[33] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer networks*, vol. 52, no. 12, pp. 2292–2330, 2008.

[34] V. Potdar, A. Sharif, and E. Chang, "Wireless sensor networks: A survey," in *Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on*, pp. 636–641, IEEE, 2009.

[35] Y. Mazzer and B. Tourancheau, "Comparisons of 6lowpan implementations on wireless sensor networks," in *Sensor Technologies and Applications, 2009. SENSORCOMM'09. Third International Conference on*, pp. 689–692, IEEE, 2009.

[36] C. Yibo, K.-M. Hou, H. Zhou, H.-l. Shi, X. Liu, X. Diao, H. Ding, J.-J. Li, and C. de Vaulx, "6lowpan stacks: a survey," in *Wireless Communications, Networking and Mobile Computing (WiCOM), 2011 7th International Conference on*, pp. 1–4, IEEE, 2011.

[37] J. J. Rodrigues and P. A. Neves, "A survey on ip-based wireless sensor network solutions," *International Journal of Communication Systems*, vol. 23, no. 8, pp. 963–981, 2010.

[38] P. Levis, "Experiences from a decade of TinyOS development," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2012), OSDI*, vol. 12, pp. 207–220, 2012.

[39] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, *et al.*, "TinyOS: An operating system for sensor networks," in *Ambient intelligence*, pp. 115–148, Springer, 2005.

[40] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-Haggerty, A. Terzis, A. Dunkels, and D. Culler, "Contikirpl and tinyrpl: Happy together," in *Workshop on Extending the Internet to Low Power and Lossy Networks (IP+ SN)*, 2011.

[41] Z. Shelby, K. Hartke, C. Bormann, and B. Frank, "Constrained Application Protocol (CoAP)," Internet-Draft draft-ietf-core-coap-18, CoRE Working Group, June 2013.

[42] A. Dunkels, "Contiki: Contiki: Bringing IP to Sensor Networks," *ERCIM News*, no. 76, 2009. [Online; accessed 07-January-2014] `http://ercim-news.ercim.eu/en76/rd/contiki-bringing-ip-to-sensor-networks`.

[43] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, pp. 455–462, IEEE, 2004.

[44] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, pp. 29–42, Acm, 2006.

[45] A. Stan, "Porting the Core of the Contiki operating system to the TelosB and MicaZ platforms," *Bachelor Thesis Computer Science, International University, Bremen, Germany*, 2007.

[46] N. Tsiftes, J. Eriksson, and A. Dunkels, "Low-power wireless IPv6 routing with ContikiRPL," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pp. 406–407, ACM, 2010.

[47] A. Dunkels, "Report: SICSlowpan-Internet-connectivity for Low-power Radio Systems." [Online; accessed 07-January-2014] `https://www.iis.se/docs/SICS_Lowpan-report.pdf`.

[48] M. Kovatsch, S. Duquennoy, and A. Dunkels, "A low-power coap for contiki," in *Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on*, pp. 855–860, IEEE, 2011.

[49] C. Tille, "Integration und Test von IPv6 für WSN (6LoWPAN, RFC4944)," Master's thesis, Universität zu Lübeck, July 2010.

[50] I. Chakeres and C. Perkins, "Dynamic MANET On-demand (DYMO) Routing," Internet-Draft draft-ietf-manet-dymo-21, Mobile Ad hoc Networks Working Group, July 2010.

[51] E. K. Kim, G. Montenegro, E. S. Park, I. Chakeres, and C. Perkins, "Dynamic MANET On-demand for 6LoWPAN (DYMO-low) Routing," Internet-Draft draft-montenegro-6lowpan-dymo-low-routing-03, Internet Engineering Task Force, Dec. 2007.

[52] SmartAssist Projektseite, "`https://www.itm.uni-luebeck.de/research/projects/smartassist/`." [Online; accessed 07-January-2014].

[53] FleGSens Projektseite, "`https://www.itm.uni-luebeck.de/research/`

`projects/flegsens`." [Online; accessed 07-January-2014].

[54] ZigBee Alliance, "The ZigBee 1.0 specification." http://www.zigbee.org/.

[55] M. Younis and K. Akkaya, "Strategies and techniques for node placement in wireless sensor networks: A survey," *Ad Hoc Networks*, vol. 6, no. 4, pp. 621 – 655, 2008.

[56] G.-H. Lin and G. Xue, "Steiner tree problem with minimum number of steiner points and bounded edge-length," *Inf. Process. Lett.*, vol. 69, pp. 53–57, Jan. 1999.

[57] J. Tang, B. Hao, and A. Sen, "Relay node placement in large scale wireless sensor networks," *Computer communications*, vol. 29, no. 4, pp. 490–501, 2006.

[58] A. A. Abbasi and M. Younis, "A survey on clustering algorithms for wireless sensor networks," *Comput. Commun.*, vol. 30, pp. 2826–2841, Oct. 2007.

[59] N. Vlajic and D. Xia, "Wireless sensor networks: to cluster or not to cluster?," in *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, pp. 258–268, IEEE Computer Society, 2006.

[60] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan, "Energy-efficient communication protocol for wireless microsensor networks," in *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*, pp. 10–pp, IEEE, 2000.

[61] V. Loscri, G. Morabito, and S. Marano, "A two-levels hierarchy for low-energy adaptive clustering hierarchy (tl-leach)," in *IEEE Vehicular Technology Conference*, vol. 62.3, p. 1809, IEEE; 1999, 2005.

[62] M. R. Ahmad, E. Dutkiewicz, and X. Huang, "Performance evaluation of MAC protocols for cooperative MIMO transmissions in sensor networks," in *Proceedings of the 5th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, PE-WASUN '08, (New York, NY, USA), pp. 54–62, ACM, 2008.

[63] N. Dimokas, D. Katsaros, L. Tassiulas, and Y. Manolopoulos, "High performance, low complexity cooperative caching for wireless sensor networks," *Wireless Networks*, vol. 17, pp. 717–737, April 2011.

[64] Y. Zeng, S. Zhang, S. Guo, and X. Li, "Secure hop-count based localization in wireless sensor networks," in *Computational Intelligence and Security, 2007 International Conference on*, pp. 907–911, IEEE, 2007.

[65] G. Werner-Allen, P. Swieskowski, and M. Welsh, "Motelab: A wireless sensor network testbed," in *Proceedings of the 4th international symposium on Information processing in sensor networks*, p. 68, IEEE Press, 2005.

[66] V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "Twist: a scalable

and reconfigurable testbed for wireless indoor experiments with sensor networks," in *Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality*, pp. 63–70, ACM, 2006.

[67] J. Beutel, R. Lim, A. Meier, L. Thiele, C. Walser, M. Woehrle, and M. Yuecel, "The flocklab testbed architecture," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pp. 415–416, ACM, 2009.

[68] CONET Testbed Directory, "`http://www.cooperating-objects.eu/testbed-simulation/testbed-federation/testbed-directory/`." [Online; accessed 10-January-2014].

[69] I. Chatzigiannakis, C. Koninis, G. Mylonas, S. Fischer, and D. Pfisterer, "WISEBED: An Open Large-Scale Wireless Sensor Network Testbed," in *Proceedings of the 1st International Conference on Sensor Networks Applications, Experimentation and Logistics*, Sept. 2009.

[70] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks," in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pp. 263–270, ACM, 1999.

[71] Hui, Jonathan W. and Culler, David E., "IP is dead, long live IP for wireless sensor networks," in *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, (New York, NY, USA), pp. 15–28, ACM, 2008.

[72] M. G. Naugle, *Network Protocol Handbook*. McGraw-Hill, Inc., 1998.

[73] G. S. Sidhu, R. F. Andrews, and A. B. Oppenheimer, *Inside AppleTalk*. Addison-Wesley Reading (Ma) etc, 1990.

[74] J. E. White, "A High-Level Framework for Network-Based Resource Sharing," RFC 707, Stanford Research Institute, Jan. 1976.

[75] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[76] R. Thurlow, "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 5531, Network Working Group, May 2009.

[77] OMG Group: Documents Associated with CORBA, 3.3, "`http://www.omg.org/spec/CORBA/3.3/`." [Online; accessed 07-January-2014].

[78] ORACLE: Java RMI, "`http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html`." [Online; accessed 07-January-2014].

[79] T. May, S. Dunning, and J. Hallstrom, "An RPC design for wireless sensor networks," *IEEE International Conference on Mobile Adhoc and Sensor Systems Conference*, vol. 0, pp. 8–138, 2005.

[80] P. C. Chapin and C. Skalka, "SpartanRPC: Secure WSN middleware for cooperating domains," in *IEEE 7th International Conference on Mobile Adhoc and Sensor Systems (MASS), 2010*, pp. 61–70, 2010.

[81] Texas Instruments - MSP430 Overview, "`http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/overview.page`." [Online; accessed 14-October-2011].

[82] Digi-Key Corporation, Search-term: JN51xx, "`http://search.digikey.com/scripts/DkSearch/dksus.dll?keywords=JN51`." [Online; accessed 14-October-2011].

[83] D. P. Agrawal, R. Biswas, A. Gupta, N. Jain, A. Mukherjee, and S. Sekhar, "Wireless sensor networks (wsns): Characteristics and types of sensors," in *Encyclopedia of Wireless and Mobile Communications*, pp. 1505–1508, Taylor & Francis, 2007.

[84] J. Arkko, H. Rissanen, S. Loreto, Z. Turanyi, and O. Novo, "Implementing Tiny COAP Sensors," Internet-Draft draft-arkko-core-sleepy-sensors-01, Network Working Group, July 2011.

[85] Embedded devices on the Internet of Things, "`http://labs.ericsson.com/system/storage/serve/512/EmbeddedDevicesOnIoT.pdf`." [Online; accessed 07-January-2014].

[86] R. Hinden and S. Deering, "IP Version 6 Addressing Architecture," RFC 4291, Network Working Group, Feb. 2006.

[87] S. Li, J. Hoebeke, F. V. den Abeele, and A. Jara, "Conditional observe in CoAP," Internet-Draft draft-li-core-conditional-observe-04, Network Working Group, June 2013.

[88] V. Srivastava and M. Motani, "Cross-layer design: a survey and the road ahead," *Communications Magazine, IEEE*, vol. 43, no. 12, pp. 112–119, 2005.

[89] S. Shakkottai, T. S. Rappaport, and P. C. Karlsson, "Cross-layer design for wireless networks," *Communications Magazine, IEEE*, vol. 41, no. 10, pp. 74–80, 2003.

[90] V. Kawadia and P. Kumar, "A cautionary perspective on cross-layer design," *Wireless Communications, IEEE*, vol. 12, no. 1, pp. 3–11, 2005.

[91] G. Holland, N. Vaidya, and P. Bahl, "A rate-adaptive mac protocol for multi-hop wireless networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, pp. 236–251, ACM, 2001.

[92] D. Kliazovich and F. Granelli, "Distributed Protocol Stacks: A Framework for Balancing Interoperability and Optimization," in *Proceedings of the IEEE International Communications Workshops, 2008*, ICC Workshops '08, pp. 241–245, IEEE Press, 2008.

[93] K. Ramakrishnan, S. Floyd, D. Black, *et al.*, "The addition of explicit

congestion notification (ecn) to ip," 2001.

[94] D. Kliazovich, N. B. Halima, and F. Granelli, "Cross-layer error recovery optimization in wifi networks," in *Wireless Communications 2007 CNIT Thyrrenian Symposium*, pp. 213–222, Springer, 2007.

[95] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed diffusion for wireless sensor networking," *IEEE/ACM Transactions on Networking (TON)*, vol. 11, no. 1, pp. 2–16, 2003.

[96] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks," in *Fifth International Conference on Information Processing in Sensor Networks*, 2006.

[97] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pp. 121–132, IEEE, 2005.

[98] P. Corke and P. Sikka, "Demo abstract : Fos —- a new operating system for sensor networks," in *5th European Conference on Wireless Sensor Networks (EWSN 2008)*, (Hotel Royal Carlton, Bologna, Italy), January 2008.

[99] W. Hu, P. Corke, W. S. Shi, and L. Overs, "secFleck: A Public Key Technology Platform for Wireless Sensor Networks," in *6th European Conference on Wireless Sensor Networks (EWSN)*, (Corke, Ireland), pp. 296–311, February 2009.

[100] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[101] D. J. Wheeler and R. M. Needham, "Tea extensions," *Unpublished manuscript, Computer Laboratory, Cambridge University, England*, 1997.

[102] P. Rothenpieler, "Poster Abstract: Distributed Protocol Stacks for Wireless Sensor Networks," in *9th European Conference on Wireless Sensor Networks (EWSN 2012)*, (Trento, Italy), 2 2012.

[103] P. Rothenpieler and D. Pfisterer, "Introduction to distributed protocol stacks for wireless sensor networks," tech. rep., 11. Fachgespräche Drahtlose Sensornetze, GI/ITG KuVS, 2012.

[104] P. Rothenpieler and D. Pfisterer, "Towards Distributed Protocol Stacks for Wireless Sensor Networks," in *Proceedings of the IEEE International Conference on Cyber, Physical and Social Computing (CPSCom 2012)*, (Besançon, France), pp. 418–425, November 2012.

[105] P. Rothenpieler, "Reliability Extensions and Multi-Hop Evaluation of Dis-

tributed Protocol Stacks," in *Proceedings of the 2013 IEEE International Conference on Cyber, Physical and Social Computing (CPSCom 2013)*, (Beijing, China), pp. 931–938, August 2013.

[106] T. Baumgartner, I. Chatzigiannakis, M. Danckwardt, C. Koninis, A. Kröller, G. Mylonas, D. Pfisterer, and B. Porter, *Virtualising Testbeds to Support Large-Scale Reconfigurable Experimental Facilities*, pp. 210–223. Springer, Heidelberg, 2010.

[107] P. Rothenpieler, D. Krüger, D. Pfisterer, S. Fischer, D. Dudek, C. Haas, A. Kuntz, and M. Zitterbart, "Flegsens - secure area monitoring using wireless sensor networks," in *Proceedings of the International Conference on Sensor Networks, Information, and Ubiquitous Computing (ICSNIUC 2009), Singapore*, vol. 56, pp. 81–92, World Academy of Science, Engineering and Technology, 2009.

[108] B. B. et.al., "Assigned Internet Protocol Numbers," tech. rep., Internet Assigned Numbers Authority, Feb. 2013.

[109] D. Dolev and A. Yao, "On the security of public key protocols," *Information Theory, IEEE Transactions on*, vol. 29, no. 2, pp. 198–208, 1983.

[110] D. Krüger, C. Haas, P. Rothenpieler, D. Dudek, and D. Pfisterer, "Security in Border Control and Area Monitoring," *it - Information Technology, Themenheft Sicherheit in Sensornetzwerken*, vol. 52, no. 6, pp. 340–344, 2010.

[111] R. Anderson, H. Chan, and A. Perrig, "Key infection: Smart trust for smart dust," in *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pp. 206–215, IEEE, 2004.

[112] M. Ramkumar and N. D. Memon, "Harps: Hashed random preloaded subset key distribution.," *IACR Cryptology ePrint Archive*, vol. 2003, p. 170, 2003.

[113] C. Karlof, N. Sastry, and D. Wagner, "Tinysec: a link layer security architecture for wireless sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 162–175, ACM, 2004.

[114] C. H. Lin, Y. Xie, and W. Wolf, "Lzw-based code compression for vliw embedded systems," in *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, vol. 3, pp. 76–81, IEEE, 2004.

[115] A. S. Tanenbaum, *Computer Networks, 4. Auflage.* Pearson Education, 2002.

[116] D. Whiting, R. Housley, and N. Ferguson, "Counter with CBC-MAC (CCM)," RFC 3610, Network Working Group, Sept. 2003.

[117] M. Bellare, J. Kilian, and P. Rogaway, "The security of the cipher block chaining message authentication code," *J. Comput. Syst. Sci.*, vol. 61,

pp. 362–399, Dec. 2000.

[118] M. Luk, G. Mezzour, A. Perrig, and V. Gligor, "Minisec: a secure sensor network communication architecture," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, pp. 479–488, IEEE, 2007.

[119] D. E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[120] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 3 ed., 2007.

[121] A. Francillon and C. Castelluccia, "Tinyrng: A cryptographic random number generator for wireless sensors network nodes," in *Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops, 2007. WiOpt 2007. 5th International Symposium on*, pp. 1–7, IEEE, 2007.

[122] J. Jonsson, "On the security of ctr+ cbc-mac," in *Selected Areas in Cryptography*, pp. 76–93, Springer, 2003.

[123] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator," in *Selected Areas in Cryptography*, pp. 13–33, Springer, 2000.

[124] K. C. Barr and K. Asanović, "Energy-aware lossless data compression," *ACM Transactions on Computer Systems (TOCS)*, vol. 24, no. 3, pp. 250–291, 2006.

[125] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, pp. 265–278, ACM, 2006.

[126] H. Lekatsas, J. Henkel, and V. Jakkula, "Design of an one-cycle decompression hardware for performance increase in embedded systems," in *Proceedings of the 39th annual Design Automation Conference*, pp. 34–39, ACM, 2002.

[127] G. T. Department, "GRAPHICS INTERCHANGE FORMAT(sm) Version 89a," tech. rep., CompuServe Incorporated, July 1990.

[128] S. Thomson, T. Narten, and T. Jinmei, "IPv6 Stateless Address Autoconfiguration," RFC 4862, Network Working Group, Sept. 2007.

[129] R. Droms, J. Bound, B. Volz, T. Lemon, C. Perkins, and M. Carney, "Dynamic Host Configuration Protocol for IPv6 (DHCPv6)," RFC 3315, Network Working Group, July 2003.

[130] C. Perkins, E. Belding-Royer, and S. Das, "Ad hoc On-Demand Distance Vector (AODV) Routing," RFC 3561, Network Working Group, July 2003.

[131] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris, "A high-throughput path metric for multi-hop wireless routing," *Wireless Networks - Special issue: Selected papers from ACM MobiCom 2003*, vol. 11, pp. 419–434, July 2005.

[132] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, (New York, NY, USA), pp. 1–14, ACM, 2009.

[133] J. Vasseur, M. Kim, K. Pister, N. Dejean, and D. Barthel, "Routing Metrics Used for Path Calculation in Low-Power and Lossy Networks," RFC 6551, Internet Engineering Task Force (IETF), Mar. 2012.

[134] J. Ko, J. Eriksson, N. Tsiftes, S. Dawson-haggerty, M. Durvy, A. Terzis, A. Dunkels, and D. Culler, "Beyond Interoperability: Pushing the Performance of Sensornet IP Stacks," in *In Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys*, 2011.

[135] Prof. Dr. Burkhard Monien, "Minimale Spannbäume," *Vorlesung: Datenstrukturen und Algorithmen, Universität Paderborn*, June 2005.

[136] D. Oran, "OSI IS-IS Intra-domain Routing Protocol," RFC 1142, Network Working Group, Feb. 1990.

[137] J. Moy, "OSPF Version 2," RFC 2328, Network Working Group, Apr. 1998.

[138] L. Zhang, W. Yang, W. N. Qian Rao, and D. Dong, "An energy saving routing algorithm based on dijkstra in wireless sensor networks," *Journal of Information & Computational Science*, May 2013.

[139] B. Musznicki, M. Tomczak, and P. Zwierzykowski, "Dijkstra-based localized multicast routing in wireless sensor networks," in *Communication Systems, Networks & Digital Signal Processing (CSNDSP), 2012 8th International Symposium on*, pp. 1–6, IEEE, 2012.

[140] K. Akkaya and M. Younis, "An energy-aware qos routing protocol for wireless sensor networks," in *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pp. 710–715, IEEE, 2003.

[141] M. Aissa, A. Ben Mnaouer, R. Murray, and A. Belghith, "New strategies and extensions in kruskals algorithm in multicast routing," *International Journal of Business Data Communications and Networking (IJBDCN)*, vol. 7, no. 4, pp. 32–51, 2011.

[142] M. X. Cheng, J. Sun, M. Min, and D.-Z. Du, "Energy-efficient broadcast and multicast routing in ad hoc wireless networks," in *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, pp. 87–94, IEEE, 2003.

[143] T. Baumgartner, I. Chatzigiannakis, S. P. Fekete, C. Koninis, A. Kröller, and A. Pyrgelis, "Wiselib: A generic algorithm library for heterogeneous sensor networks," *CoRR*, vol. abs/1101.3067, 2011.

[144] D. Stirpe, "IPv6 Routing Protocol for Low-Power and Lossy Networks (RPL): Implementation and Evaluation over Wisebed," Master's thesis, Sapienza University of Rome, July 2013.

[145] I. Free Software Foundation, "Gnu general public license, version 3." `https://www.gnu.org/licenses/gpl.html`, June 2007. Last retrieved 2013-09-13.

[146] D. Gehberger, "IPv6 protocol stack implementation with 6LoWPAN support for Wiselib," Master's thesis, Budapest University of Technology and Economics, Dec. 2012.

[147] P. Rothenpieler and D. Pfisterer, "Introduction to distributed protocol stacks for wireless sensor networks," tech. rep., 11. Fachgespräche Drahtlose Sensornetze, GI/ITG KuVS, Darmstadt, Germany, 2012. to appear.

[148] X. Han, X. Cao, E. Lloyd, and C.-C. Shen, "Fault-tolerant relay node placement in heterogeneous wireless sensor networks," in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pp. 1667–1675, 2007.

[149] H. Liu, P. jun Wan, and X. Jia, "Fault-tolerant relay node placement in wireless sensor networks," *LNCS*, vol. 3595, pp. 230–239, 2005.

[150] E. Lloyd and G. Xue, "Relay node placement in wireless sensor networks," *Computers, IEEE Transactions on*, vol. 56, no. 1, pp. 134–138, 2007.

[151] V. K. Wan and K. D. Ba, *CS 105 Algorithms (Graduate Level): Approximation Algorithms*. Department of Computer Science, Dartmouth College, 2005.

[152] M. Ishizuka and M. Aida, "Performance study of node placement in sensor networks," in *Distributed Computing Systems Workshops, 2004. Proceedings. 24th International Conference on*, pp. 598–603, 2004.

[153] N. Bulusu, D. Estrin, L. Girod, and J. Heidemann, "Scalable coordination for wireless sensor networks: self-configuring localization systems," in *in Proc. 6th International Symposium on Communication Theory and Applications (ISCTA 01),Ambleside, Lake District*, 2001.

[154] X. Wang, G. Xing, Y. Zhang, C. Lu, R. Pless, and C. Gill, "Integrated coverage and connectivity configuration in wireless sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, pp. 28–39, ACM, 2003.

[155] F. V. Conesa, "Avaluacio d'una implementacio de 6LoWPAN: el cami a la Internet de les coses," Master's thesis, Escola Politecnica Superior de Castelldefels, 2009.

[156] A. Ludovici, A. Calveras, and J. Casademont, "Forwarding Techniques for IP Fragmented Packets in a Real 6LoWPAN Network," *Sensors 2011*, 2011.

[157] B. Cody-Kenny, D. Guerin, D. Ennis, R. Simon Carbajo, M. Huggard, and C. Mc Goldrick, "Performance evaluation of the 6lowpan protocol on micaz and telosb motes," in *Proceedings of the 4th ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, PM2HW2N '09, (New York, NY, USA), pp. 25–30, ACM, 2009.

[158] B. Latré, P. De Mil, I. Moerman, N. Van Dierdonck, B. Dhoedt, and P. Demeester, "Maximum throughput and minimum delay in ieee 802.15. 4," in *Mobile Ad-hoc and Sensor Networks*, pp. 866–876, Springer, 2005.

[159] M. Afanasyev, D. O'Rourke, B. Kusy, and W. Hu, "Heterogeneous traffic performance comparison for 6lowpan enabled low-power transceivers," in *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, HotEmNets '10, (New York, NY, USA), pp. 10:1–10:5, ACM, 2010.

[160] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text compression*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[161] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Comput. Surv.*, vol. 21, pp. 557–591, Dec. 1989.

[162] P. Zhang, C. M. Sadler, S. A. Lyon, and M. Martonosi, "Hardware design experiences in zebranet," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 227–238, ACM, 2004.

[163] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An analysis of a large scale habitat monitoring application," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pp. 214–226, ACM, 2004.

[164] A. Guinard, A. McGibney, and D. Pesch, "A wireless sensor network design tool to support building energy management," in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '09, (New York, NY, USA), pp. 25–30, ACM, 2009.

[165] X. Jiang, M. Van Ly, J. Taneja, P. Dutta, and D. Culler, "Experiences with a high-fidelity wireless building energy auditing network," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, (New York, NY, USA), pp. 113–126, ACM, 2009.

[166] M. Chetty, D. Tran, and R. E. Grinter, "Getting to green: understanding resource consumption in the home," in *Proceedings of the 10th international conference on Ubiquitous computing*, UbiComp '08, (New York, NY, USA), pp. 242–251, ACM, 2008.

[167] J. K. Dobson and A. J. D. Griffin, "Conservation effect of immediate

electricity cost feedback on residential consumption behaviour," in *American Council for an Energy-Efficient Economy: Summer Study on Energy Efficiency in Buildings*, 1992.

[168] D. T. Delaney, G. M. P. O'Hare, and A. G. Ruzzelli, "Evaluation of energy-efficiency in lighting systems using sensor networks," in *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '09, (New York, NY, USA), pp. 61–66, ACM, 2009.

[169] C. Gomez, A. Boix, and J. Paradells, "Impact of lqi-based routing metrics on the performance of a one-to-one routing protocol for ieee 802.15.4 multihop networks," *EURASIP J. Wirel. Commun. Netw.*, vol. 2010, pp. 6:1–6:20, Feb. 2010.