Dienstorientierte Sensornetze: Programmierabstraktionen und Dienstkomposition

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Bernhard Alexander Hurler

aus Tettnang

Tag der mündlichen Prüfung: 10. Februar 2012

Erste Gutachterin: Prof. Dr. Martina Zitterbart

Zweiter Gutachter: Prof. Dr. Ralf Reussner

Für
Antonia und Thalía

Vorwort

Die vorliegende Arbeit entstand während meiner Zeit als wissenschaftlicher Mitarbeiter am Institut für Telematik des Karlsruher Instituts für Technologie (KIT). Frau Prof. Dr. Martina Zitterbart danke ich sehr, dass sie mir die Möglichkeit zur Promotion gegeben hat und mich während der ganzen Zeit mit Ansporn und Geduld in der jeweils richtigen Dosierung begleitet hat. An ihrem Institut forschen zu dürfen, war Herausforderung und Ehre, auch hier in einer gelungenen Dosierung. Mein Dank gilt auch Herrn Prof. Dr. Reussner für die Übernahme des Korreferats.

Ich möchte mich ganz herzlich bei meinen Sensorkollegen am Institut für Telematik bedanken: Prof. Dr. Erik-Oliver Blaß und Prof. Dr. Achim Hof, mit denen ich die Anfänge der Sensornetzforschung am Institut für Telematik mitgestalten durfte, und Detlev Meier, ohne dessen profundes Elektronikwissen es nie einen Sensornetz-Demonstrator mit Gartenhaus gegeben hätte. Zudem möchte ich mich bei Dr. Uwe Walter bedanken, der sich als wertvoller Begleiter in promotionstechnisch harten Zeiten bewährt hat.

Zahlreiche Studenten haben zum Gelingen dieser Arbeit beigetragen, wofür ich mich bei allen herzlich bedanke. Unter den vielen möchte ich Sebastian Mies, Anton Hergenröder und Christian Haas herausheben, die die selbstpflegenden Pflanzen im Sensornetz von der ersten Stunde an begleitet haben.

Ganz herzlich möchte ich meinen Eltern, meiner Frau, meinen Kindern und meinen Geschwistern danken, die alle - jeder auf seine Weise - sichtbare Spuren in dieser Arbeit hinterlassen haben. Ich danke Euch sehr!

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung und Problemstellung

Die moderne Menschheit ist auf umfassende Informationen angewiesen. Gesellschaftswissenschaften und Medien prophezeien deswegen nach dem Übergang von der Agrar- über die Industrie- zur Dienstleistungsgesellschaft den Wandel zur Informationsgesellschaft. Mit dem Bedürfnis nach Information wächst gleichzeitig das Bedürfnis nach Daten, die stets Grundlage jeder Information sind. Auf dem Feld der Datenerhebung hat das Zusammentreffen mehrerer technischer Entwicklungen - Miniaturisierung drahtloser Kommunikationsmodule, gesteigerte Leistungsfähigkeit und verminderte Energieaufnahme von Mikroprozessoren, Verkleinerung und Steigerung der Kapazität von Batterien und Akkumulatoren - ein neues wissenschaftliches Forschungsgebiet entstehen lassen: drahtlose Sensornetze.

Einige der Ideen und Ergebnisse dieses Forschungsgebiets haben bereits den Weg in die Wirtschaft und die populäre Presse gefunden. So stellt beispielsweise Nokia ein Handy vor, das sowohl Umweltdaten wie auch Körperdaten drahtlos erfasst [Nok08]. Auch in der Kleidung finden sich bereits Sensorknoten, die Sportlern und Patienten eine drahtlose Langzeitbeobachtung von Körperdaten ermöglichen [Deu10]. Der Deutschlandfunk berichtet über Schwärme von Sensorknoten, die nicht nur ihre Umgebung mit Sensoren physikalisch vermessen, sondern sich auch autonom an verschiedene Situationen anpassen können [Deu07]. Sensorbestückte Handys bilden spontan und flexibel ein dichtes Sicherheitsnetz zur Überwachung [SO09] und weisen auf sichere Fluchtrichtungen und -wege hin, indem sie aktuelle Daten sammeln, verbreiten und auswerten [Hei10].

Drahtlose Sensornetze bestehen aus Sensorknoten, die ihre Umwelt durch physikalische Sensoren messen und wahrnehmen. Sind zusätzlich physikalische Aktoren vorhanden, die ihre Umwelt beeinflussen können, spricht man von Aktorknoten und damit von Sensor-Aktor-Netzen. Im Folgenden wird nicht speziell zwischen Sensoren und Aktoren unterschieden, weswegen der Begriff Sensorknoten verallgemeinert auch für Sensor-Aktor-Knoten gilt, ebenso Sensornetz für Sensor-Aktor-Netz.

Sensorknoten sind autonome Einheiten, die drahtlos über Funk kommunizieren können. Zudem besitzen Sensorknoten eine eigene Recheneinheit (CPU) und eine eigene Energieversorgung durch Batterien oder Fähigkeiten zur Energiegewinnung.

Besondere Herausforderungen bei Sensornetzen

Drahtlose Sensornetze können komplexe Aufgaben übernehmen, wie zum Beispiel Beobachtung von Umweltdaten in Katastrophenszenarien, Überwachung und Langzeitspeicherung von Körperfunktionen in der Medizin, und Regelungsaufgaben in privaten oder professionellen Umgebungen. Eine besondere Herausforderung dieser Sensornetze ist die Ressourcenbeschränktheit der einzelnen Sensorknoten, was ihre Rechenkapazität, ihre Speicherkapazität, die Kommunikationsfähigkeit und ihre Energiekapazität anbelangt. Während die Ressourcen aller Sensorknoten in Summe ausreichend für die vorgenannten Aufgaben sind, ist die Leistungsfähigkeit eines einzelnen Sensorknotens äußerst limitiert. Insbesondere wenn sich die Anforderungen an eine Sensornetz-Anwendung während der Laufzeit ändern (z. B. energiesparende Kommunikation im Normalbetrieb, schnelle Kommunikation in Alarmsituationen), sind diese Limitierungen zum Zeitpunkt der Programmierung (statisch) und zum Zeitpunkt der Ausführung (dynamisch) zu beachten.

Sensornetze werden für sehr unterschiedliche Anwendungsfälle genutzt, und innerhalb dieser Anwendungen sind die Anforderungen an die Kommunikation sehr verschieden, unter Umständen sogar wechselnd. Allgemeingültige Programmierabstraktionen und Kommunikationsprotokolle sind deswegen äußerst schwer zu finden, weshalb die meisten Sensornetze für einen speziellen Anwendungsfall implementiert und während der Lebenszeit des Sensornetzes nicht mehr geändert werden. In gleicher Weise sind Kommunikationsprotokolle auf spezielle Umgebungen und Anwendungsanforderungen optimiert.

Mit der zunehmenden Existenz realer, bereits ausgebrachter Sensornetze besteht jedoch dringend der Bedarf, Sensornetze für mehrere Anwendungen zu nutzen, und diese während ihrer Lebenszeit zu ändern oder sie auf veränderte Bedingungen anzupassen. Ebenso soll die Kommunikation innerhalb eines Sensornetzes nicht auf a-priori festgelegte Optimierungsziele festgelegt sein, sondern sich den Umweltbedingungen und den wechselnden Anforderungen einer Anwendung anpassen lassen. Mit den bestehenden Ansätzen können jedoch diese Anforderungen nicht erfüllt werden.

Zur Lösung dieses Problems werden in dieser Arbeit Programmierabstraktionen entwickelt, die eine flexible Nutzung von Sensornetzen mit wechselnden Anwendungen unter wechselnden Kommunikationsanforderungen erlauben. Eine Programmierabstraktion zur Implementierung von Sensornetzanwendungen muss insbesondere der inhärent verteilten Natur dieser Anwendungen gewachsen sein. Ein Sensornetz besteht aus vielen Sensorknoten, auf denen jeweils ein Teil der übergeordneten Anwendung abläuft. Die Verteilung der jeweiligen Anwendungsteile auf die Sensorknoten beeinflusst wesentlich den Kommunikationsaufwand. Da Sensorknoten nur geringe Energie besitzen, vermindert ein hoher Kommunikationsaufwand und damit ein hoher Energieverbrauch die Lebenszeit eines Sensorknotens beträchtlich.

Bei einer großen Anzahl von Knoten ist es jedoch dem Programmierer nicht möglich, manuell eine günstige Verteilung auf Sensorknoten zu bestimmen. Deswegen muss auch die Programmierabstraktion diese Verteilung auf natürliche Weise unterstützen. Gleichzeitig ist es wichtig, dass die Komplexität der atomaren (also auf einem einzigen Sensorknoten ablaufenden) Anwendungsteile den begrenzten Ressourcen angepasst sind. Idealerweise sollte der durch die Programmierabstraktion vorgegebene Aufbau einer Sensornetzanwendung auch eine automatisierte Verteilung erlauben. Eine manuelle A-priori-Zuordnung der Anwendungsteile auf bestimmte Sensorknoten ist bei komplexen Sensornetzanwendungen und großer Anzahl an Sensorknoten nicht mehr möglich.

Ein analoges Bild zeigt sich im Bereich der Sensordaten selbst. Sie bilden die Basis jedes Sensornetzes und jeder Sensornetzanwendung. Während bestimmte Anwendungen mit den Rohdaten der physikalischen Sensoren auskommen, sind andere von einer netzinternen Verarbeitung jener abhängig oder profitieren davon. Deswegen muss neben der Programmierung des Kontrollflusses von Anwendungen auch eine Programmierung von anwendungsspezifischen Datentypen möglich sein. Neben der Aggregation homogener Sensordaten, z. B. durch Mittelwertbildung, sollten sich auch heterogene Daten verknüpfen lassen und Einzeldaten verändert, z. B. reduziert, werden können.

Neben Kontrollfluss und Nutzdaten spielt die drahtlose Kommunikation eine maßgebliche Rolle bei verteilten Sensornetzanwendungen. In den meisten Fällen (d. h. bei den meisten für Sensornetze entwickelten Protokollen) ist es das oberste Ziel, Energie zu sparen. Obwohl dieses Optimierungsziel für viele Situationen richtig ist, genügt es nicht permanent allen Anwendungen. Auch im Bereich der Kommunikation ist deshalb ein flexibler Ansatz erforderlich, der situations- und anwendungsbezogen verschiedene Optimierungsziele erlaubt. Das Optimierungsziel der Kommunikation muss dabei frei gewählt und gewechselt werden können.

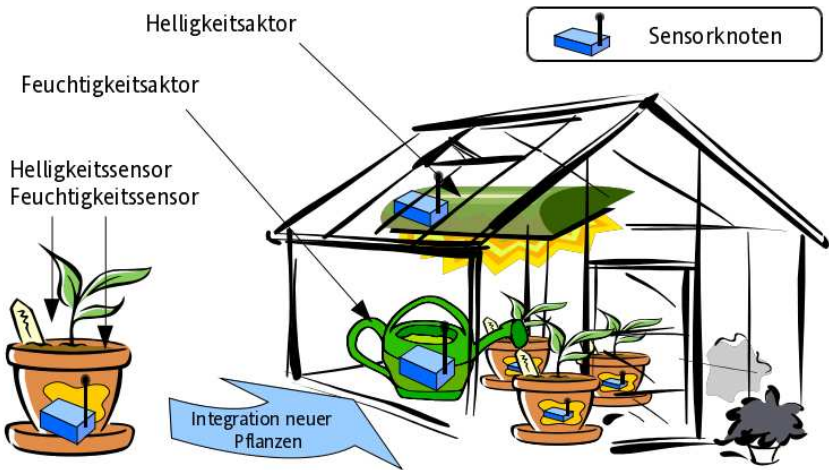

Abbildung 1.1: Sensornetzbasiertes intelligentes Gewächshaus mit TALASSA

Zusammengefasst sollen sich Sensornetze auf veränderte Bedingungen oder Anforderungen anpassen lassen. Dies umfasst die Möglichkeit zur Programmierung und Änderung von verteilten Anwendungen, Definition von anwendungsspezifischen Daten und eine Anpassbarkeit der Kommunikation.

Ziele und Inhalte der Arbeit

Das übergeordnete Ziel dieser Arbeit ist es, Sensornetze flexibel einsetzen zu können. Dies wird erreicht, indem folgende drei Problembereiche untersucht und Lösungen dafür entwickelt werden: verteilte Anwendungsprogrammierung, anwendungsspezifische Datenverarbeitung und situationsbezogene Anpassung der Kommunikation.

Zur Lösung des ersten Problembereichs wird ein dienstorientierter Ansatz zur Programmierung und Ausführung von verteilten Anwendungen entwickelt. Dieser neue dienstorientierte Ansatz stellt den Dienst als atomare kleinste Einheit in den Mittelpunkt. Ein Dienst wird immer von einem einzelnen Sensorknoten ausgeführt, wobei ein Sensorknoten auch viele dieser atomaren Dienste ausführen kann. Die Ausführungssemantik eines Dienstes wird über wenige Parameter bestimmt und geschieht mittels der in dieser Arbeit entwickelten Sprache TALASSA (TAsking LAnguage for Service-oriented Sensor Actuator networks). Insgesamt stehen zwei Dienstgruppen mit insgesamt fünf Diensttypen zur Verfügung, mit denen verteilte Anwendungen programmiert werden können.

Die erste Gruppe, die sogenannten primitiven Dienste, enthält Diensttypen, die physikalische Sensoren auslesen beziehungsweise physikalische Aktoren steuern. Primitive Dienste greifen auf die zur Verfügung stehende Hardware (Sensoren oder Aktoren) eines Sensorknoten zu. Mit ihnen lässt sich somit der Nutzdatenfluss einer verteilten Anwendung definieren. Die zweite Gruppe, die sogenannten komponierenden Dienste, enthält Diensttypen, die andere Dienste verbinden und steuern. Die komponierenden Dienste sind unabhängig von physikalischen Sensoren und Aktoren und können somit auf jedem Sensorknoten ablaufen. Mit diesen komponierenden Diensten ist es möglich, Dienste wiederholt, unter gewissen Bedingungen oder bei speziellen Ereignissen zu starten beziehungsweise zu beenden. Mit ihnen lässt sich der Kontrollfluss einer verteilten Anwendung definieren.

In dieser Arbeit wird gezeigt, dass sich mit dem entwickelten dienstorientierten Ansatz verteilte Sensornetz-Anwendungen programmieren lassen, die sich zur Laufzeit an sich ändernde Umstände bei Kommunikation und Energie selbständig anpassen können und damit den Energieverbrauch erheblich vermindern. Ebenso ermöglicht der vorgestellte Ansatz den Einsatz neuer Anwendungen und die Änderung bestehender Anwendungen, auch nachdem ein Sensornetz ausgebracht worden ist. Dies wird mit einem dienstorientiert programmierten intelligenten Gewächshaus demonstriert, wie in Abb. 1.1 dargestellt. In diesem werden Sensoren für Helligkeit und Feuchtigkeit mit ihren jeweils entsprechenden Aktoren genutzt, Pflanzen nach vorgegebenen Parametern zu versorgen. Die bestehende verteilte Anwendung zur Versorgung der Pflanzen wird durch die Integration neuer Pflanzen durch Talassa-Mechanismen automatisch so geändert, dass die Versorgungsparameter allen Pflanzen gerecht werden.

Zur Lösung des zweiten Problembereichs wird ein datenorientierter Ansatz zur netzinternen Verarbeitung von Sensordaten entwickelt. Die netzinterne Datenverarbeitung wird mittels der in dieser Arbeit entwickelten Sprache Pan (Process and Aggregate Named data) definiert. Vergleichbar mit dem sprachlichen Aufbau bei Talassa gibt es auch hier zwei Gruppen von Daten: primitive und abgeleitete Daten. Daten, die direkt von physikalischen Sensoren gelesen werden, sind in dieser Nomenklatur primitiv. Neben dem eigentlichen Sensorwert enthalten diese Daten Informationen über ihren Typ, das Alter (seit der Datenerfassung vergangene Zeit) und ihre Herkunft. Auch abgeleitete Daten enthalten diese Informationen, werden jedoch unabhängig von physikalischen Sensoren definiert. Abgeleitete Daten enthalten zusätzlich Informationen, wie der Wert aus anderen (primitiven oder abgeleiteten) Daten abgeleitet wird. Mit ihnen ist es möglich, anwendungsspezifische Daten, z. B. Aggregation durch Mittelwertbildung, zu definieren.

Zur Lösung des dritten Problembereichs wird ein modusbasierter Ansatz zur Steuerung des Kommunikationsverhaltens entwickelt. Das in dieser Arbeit entwickelte Modus-Rahmenwerk erlaubt eine anwendungsgesteuerte Festlegung des Kommunikationsverhaltens. Innerhalb des Modus-Rahmenwerks werden a-priori gewünschte Optimierungsziele der Kommunikation, sogenannte Modi, festgelegt. Ein Modus legt das Verhalten eines Knoten innerhalb dieses Modus fest. Eine spezielle Modus-Implementierung kann somit beliebige Ziele verfolgen, z. B. energiesparende Kommunikation für den Normalbetrieb, schnelle Kommunikation für Alarmsituationen, oder robuste Kommunikation für Situationen mit hoher Fehler- oder Ausfallwahrscheinlichkeit. Das Rahmenwerk bietet über eine abstrakte Schnittstelle die Möglichkeit, zwischen den einzelnen Modi netzweit und konsistent zu wechseln. Die Entscheidung über den Zielmodus und den Zeitpunkt des Wechsels obliegt dabei der Anwendung und kann von einem beliebigen Knoten ausgelöst werden.

In dieser Arbeit wird gezeigt, dass das Modus-Rahmenwerk einen konsistenten und korrekten Wechsel des Kommunikationsverhaltens eines Sensornetzes ermöglicht. Die Effizienz des Modus-Wechsels kann dabei mit der Wahl des Modus-Wechsel-Protokolls auf Schnelligkeit für den netzweiten Modus-Wechsel, Speicherplatzverbrauch oder Einfachheit der Implementierung optimiert werden.

Gliederung der Arbeit

Im folgenden zweiten Kapitel werden Grundlagen für Sensornetze dargestellt. Neben den allgemeinen Voraussetzungen und Randbedingungen in Sensornetzen werden typische Anwendungsszenarien und der Stand der Technik bei Programmierung und Kommunikationsprotokollen in Sensornetzen vorgestellt. Im dritten Kapitel wird der in dieser Arbeit entwickelte dienstorientierte Ansatz in Sensornetzen vorgestellt, der eine Entwicklung von verteilten Anwendungen und Definition anwendungsspezifischer Datenverarbeitung erlaubt. Das vierte Kapitel erläutert den in dieser Arbeit entwickelten Ansatz zur anwendungsgesteuerten Kommunikationsoptimierung, die modusbasierte Kommunikation. Abschließend werden im fünften Kapitel die Ergebnisse dieser Arbeit kurz zusammengefasst und ein Ausblick auf mögliche Weiterentwicklungen gegeben.

2 Grundlagen Sensornetze

In diesem Kapitel wird in das Gebiet der Sensornetze eingeführt. Zu Beginn werden in Abschnitt 2.1 Anwendungen für Sensornetze aufzählend dargestellt, um einen Einblick in die breiten Einsatzmöglichkeiten zu geben. Im Anschluss wird in Abschnitt 2.2 der Begriff der dienstorientierten Sensornetze eingeführt und von datenorientierten Sensornetzen abgegrenzt. Den Abschluss bildet in Abschnitt 2.3 ein Überblick über die für diese Arbeit relevanten Forschungsarbeiten im Bereich der Sensornetze.

2.1 Einsatzszenarien und Anwendungen

Generell sind Sensornetze für eine Vielzahl von Anwendungen geeignet. Zum einen sind alle Szenarien, in denen physikalische Daten in einem geographisch weit verteilten Raum zu erheben sind, für Sensornetze prädestiniert. Zum anderen existieren Szenarien, in denen neben der Messung der physikalischen Umgebung diese auch über Aktoren beeinflusst und geändert wird. Im Folgenden werden verschiedene Kategorien von Anwendungsgebieten erläutert (siehe auch [MI05] und [Tol05]).

Technischer und häuslicher Bereich: In diesem Bereich sind u. a. Anwendungen für die Wartung und Überwachung von industriellen und privaten Anlagen zu finden [Oeh10] [BK10]. Dazu zählt auch die Kontrolle und Überwachung von einzelnen Maschinen und Geräten. Im Automobilsektor beispielsweise ist seit langem die Verwendung einer immer größer werdenden Anzahl von (meist drahtgebunden kommunizierenden) Sensoren und Aktoren im Fahrzeug festzustellen. Neben abgeschlossenen Regelkreisen, in denen Sensor-Aktor-Paare unmittelbar eine physikalische Größe steuern, wie z. B. Temperatur, Licht, etc., ist auch eine Zusammenarbeit unterschiedlicher Sensoren und Aktoren möglich. So wird beispielsweise im Fahrzeug die Temperatur in Abhängigkeit der Sonneneinstrahlung, Luftqualität, -feuchtigkeit und individuellen Vorgaben in

verschiedenen Zonen unterschiedlich geregelt, um eine gesamtheitlich angenehme und sichere Reisesituation zu schaffen [Dai11].

Ein großer Anwendungsbereich stellt auch die sogenannte Heimautomatisierung dar. Ausgestattet mit Bewegungssensoren, elektronischen Türschildern, Temperatur-, Licht-, Feuchtigkeits- und weiteren Sensoren können Büroräume und private Wohnungen nach individuellen Vorgaben gesteuert und beobachtet werden [GMM+08].

Wenn nicht nur die Infrastruktur, sondern auch die Personen selbst mit Sensoren ausgestattet sind, ergeben sich weitere Anwendungsmöglichkeiten. Ermöglicht werden beispielsweise Bewegungsbeobachtungen zur Durchführung von soziologischen Studien [PGL+10], Anwendungen zur Steuerung von Gebäude- oder Stadionevakuierungen [TPT06], sowie situationsangepasste und personalisierte Steuerungen oben genannter Regelkreise.

Landwirtschaft und Umwelttechnik: Die Hauptanwendung in diesem Bereich liegt in der Überwachung geographisch großer Flächen. Mit einer potentiell hohen Anzahl von Messpunkten und ohne die Notwendigkeit zur Installation einer festen technischen Infrastruktur bieten hier drahtlose Sensornetze eine Vielzahl neuer Anwendungsmöglichkeiten. Sensornetze können eingesetzt werden, um Waldbrände zu detektieren und zu beobachten [HB07], Wasserströmungen und -temperatur in Meeren zu messen [STD+10] [ICT11], oder Daten über Gletscherbewegungen langfristig und feinmaschig zu erheben [MOH04].

Im Bereich der Biologie können Sensornetze genutzt werden, um Pflanzen und Tiere ohne die Störung von Menschen in ihrer natürlichen Umgebung zu beobachten. Eines der ersten bekannt gewordenen Forschungssensornetze diente der Beobachtung der Tierwelt auf der Great-Duck-Insel. Im Projekt „Habitat Monitoring on Great Duck Island“ (siehe [SMP+04]) wurde die dortige Tierwelt durch ein drahtloses Sensornetz über längere Zeit beobachtet und die Daten konnten in Echtzeit im Internet abgerufen werden.

Bauwesen: In diesem Bereich steht u. a. die Überwachung von Gebäuden im weitesten Sinne im Mittelpunkt. Durch den Einsatz von drahtlosen Sensornetzen können Bauwerke wie Stadien, Hochhäuser, Kraftwerke, Brücken und Tunnel langfristig und exakt untersucht werden [XRC+04] [KPC+06] [JL10]. Dadurch können die Auswirkungen von äußeren Einflüssen wie Wind, Erdbeben, Hochwasser oder Bränden beobachtet werden, um bei Bedarf akute Gegenmaßnahmen zu ergreifen oder beim Bau zukünftiger Anlagen als Datengrundlage zu dienen.

Militärischer Bereich: Sensornetze standen früh im Fokus militärischen Interesses. Schon im Jahr 2002 machte beispielsweise die DARPA (U. S. Department of Defense's Advanced Research Projects Agency) Sensortechnik und Sensornetze zum Forschungsschwerpunkt (laut [MR03]). Die DARPA förderte mit einer Summe von 160 Millionen US-Dollar direkt die Erforschung von Sensornetzen. Weitere 500 Millionen US-Dollar kamen von anderen US-Behörden dazu (laut [Ful02]).

Konkrete Anwendungen in diesem Bereich sind die Überwachung von Soldaten und Fahrzeugen im Kampfeinsatz und das Erkunden und Beobachten größerer Territorien. Sensornetze können ebenfalls für die Entwicklung neuer beziehungsweise zur Verbesserung existierender Waffen eingesetzt werden. Beispielsweise sind als Sensornetz angelegte Minenfelder denkbar, die mithilfe der gesammelten Sensordaten Kollateralschäden und „friendly fire“ infolge fälschlicher Auslösung von Explosionen vermeiden. Auch die Überwachung langer und unübersichtlicher Grenzen kann durch drahtlose Sensornetze ermöglicht oder verbessert werden [RKP+09] [DHK+09].

Medizinischer Bereich: Im medizinischen Bereich werden seit langem Körperfunktionen gemessen, um Krankheiten festzustellen und deren Verlauf zu protokollieren. Mit drahtlosen Sensornetzen vereinfacht sich diese Methode, da der Patient unabhängig von Arztbesuchen kontrolliert werden kann und somit beispielsweise auch Langzeitbeobachtungen im Alltag möglich sind [DH11].

Unmittelbar verwandt sind Anwendungen im sportmedizinischen und freizeitsportlichen Bereich. Drahtlose Sensornetze vereinfachen auch hier die Erhebung von relevanten Körperdaten wie Blutdruck, Temperatur und Puls, um den Sportler während seiner sportlichen Aktivität zu beobachten und notfalls zu warnen, oder um Trainingsfortschritte über einen langen Zeitraum zu protokollieren [JOLR+03].

2.1.1 Kollektive und kollaborative Anwendungen

Im vorhergehenden Abschnitt wurden zahlreiche Anwendungen aufgezählt und nach Anwendungsgebieten kategorisiert. Während diese Art der Kategorisierung einen guten Überblick über die vielfältigen Einsatzmöglichkeiten von Sensornetzen gewährt, erlaubt sie keine Einschätzung der Anforderungen an Kommunikation und Programmierung der Sensornetze. Eine für diese Einschätzung besser geeignete Einordnung ergibt sich, wenn die Interaktion zwischen den Sensorknoten und der Datenfluss der Sensordaten beachtet wird.

kollektiv

Viele Sensornetze beziehungsweise Sensornetzanwendungen basieren auf der Sammlung von Sensordaten. Die Sensoren dienen als *kollektive* Datenquelle und werden klassischerweise als große Datenbank betrachtet. Netzübergreifende Datenanfragen und netzinterne Datenfusion spielen hier eine große Rolle. Von den oben erwähnten Anwendungen gehören die Wartung und Überwachung von Anlagen, Umweltbeobachtung, Tierbeobachtung und die Beobachtung von Gebäuden und Bauwerken in diese Kategorie.

kollaborativ

Eine andere Gruppe von Anwendungen stellt nicht das Datensammeln in den Mittelpunkt, sondern die Erfüllung von Aufgaben. Dabei reagieren die Sensorknoten auf die Daten anderer Knoten mit vordefinierten Aktionen, wobei häufig zusätzlich Aktoren beteiligt sind, die die Umwelt aktiv beeinflussen. Die Sensorknoten arbeiten in diesen Anwendungen *kollaborativ* zusammen, um eine bestimmte Aufgabe gemeinsam zu erfüllen. In diese Kategorie fallen Anwendungsszenarien beim intelligenten Haus, die Überwachung von Grenzen, Anwendungen zur Steuerung von Gebäude- oder Stadionevakuierungen sowie situationsangepasste und personalisierte Steuerungen von Regelkreisen.

Diese Arbeit legt den Fokus hauptsächlich auf die Lösung von Herausforderungen, die bei *kollaborativen* Anwendungen auftreten, auch wenn wesentliche Teile der Herausforderung *kollektiver* Sensornetze ebenfalls beachtet werden. Im Folgenden wird deswegen ein Leitszenario aus dem Bereich der kollaborativen Sensornetze vorgestellt. Anschließend wird die Charakteristik von kollektiven und kollaborativen Sensornetze ausführlich diskutiert und die Verbindung zu daten- und dienstorientierten Sensornetzen aufgezeigt.

2.1.2 Leitszenario: Intelligentes Gewächshaus

In dieser Arbeit stehen Abstraktionen zur Programmierung von Sensornetzen im Vordergrund. Hier wird ein Szenario für ein konkretes Sensornetz vorgestellt, anhand dem später die entwickelten Architekturen, Modelle und Sprachkonstrukte verdeutlicht werden können.

Der Rahmen für das Szenario ist ein mit Pflanzen bestücktes Gewächshaus. Die einzelnen Pflanzentöpfe sind mit Sensorknoten ausgestattet, die eine Feuchtigkeitsmessung der Pflanzenerde erlauben. Jeder Pflanze steht außerdem eine individuelle Wasserversorgung mit steuerbarer Wasserzufuhr zur Verfügung. Die Helligkeit im Gewächshaus wird durch einen Lichtsensor gemessen. Durch eine Pflanzenlampe kann die Helligkeit im Gewächshaus verändert werden. Mit den zur Verfügung stehenden Sensoren und

Aktoren soll jede Pflanze nach einer pflanzenindividuellen Pflegeanleitung optimal mit Wasser und Licht versorgt werden. Die Feuchtigkeit in den Pflanzengefäßen kann separat gemessen und gesteuert werden, das Licht hingegen gilt übergreifend für alle Pflanzen und wird deswegen für alle Pflanzen im Gewächshaus gemeinsam geregelt. Neue Pflanzen können dem intelligenten Gewächshaus jederzeit hinzugefügt werden, wobei das Sensornetz die, durch die hinzugefügte Pflanze, neuen Bedingungen erfasst und Regelungen entsprechend anpasst.

Dieses Szenario dient vor allem in Kapitel 3 als Rahmen für einzelne Beispiele. Nach Bedarf wird es verfeinert und dient insbesondere bei der Evaluation des in dieser Arbeit entwickelten Ansatzes eines dienstorientierten Sensornetzes als Grundlage.

2.2 Datenorientierte und dienstorientierte Sensornetze

Drahtlose Sensornetze bilden seit einigen Jahren einen neuen Forschungsschwerpunkt im Bereich der drahtlosen Netze. Ihren forschungsrelevanten Ursprung finden sie in den drahtlosen, mobilen Ad-hoc-Netzen (MANET für engl. *mobile ad-hoc networks*). Die Herausforderung in diesem Forschungsbereich lag und liegt in der Nutzung von drahtlos kommunizierenden und sich bewegenden Endgeräten, wie z. B. Mobilfunkgeräten, Laptops oder PDAs (für engl. *Personal Digital Assistant*). Es wurden neue Protokolle entwickelt, die den besonderen Eigenschaften der drahtlosen Kommunikation (stochastische und systematische Fehleranfälligkeit) und der sich bewegenden Geräte (spontaner Ausfall und Eintritt, spontane Änderung der Netztopologie) Rechnung tragen. Im Gegensatz zu den drahtgebundenen Netzen wird nicht davon ausgegangen, dass Verbindungen zwischen End- oder Zwischengeräten in ihrer Qualität und ihrer Dauer *stabil* sind, sondern durch stochastische, nicht-berechenbare Einflüsse immanent *instabil* sind. Die darüberliegenden Anwendungen auf den Endgeräten sollen dabei weitgehend unberührt von den Störeinflüssen der instabilen Kommunikation ablaufen. Wenngleich sich die Kommunikation in MANETs fundamental von drahtgebundenen Netzen unterscheidet, bleiben wichtige Paradigmen dieser Netze bezüglich der Kommunikation und Anwendung erhalten: Sowohl Anwendungen wie auch die Kommunikation sind Endgerät-bezogen, d. h. eine Anwendung wird von einem Benutzer auf einem *bestimmten Gerät* betrieben, die wiederum mit Anwendungen auf einem oder mehreren *bestimmten Geräten* kommuniziert. Dieses *bestimmte* Gerät wird dabei durch eine eindeutige Netz-Adresse angegeben. Dies bedeutet insbesondere, dass Anwendungen Endpunkte

der Kommunikation darstellen und der Benutzer *seine* Anwendung (auch wenn diese netzweit betrachtet ein Teilnehmer eines Anwendungsverbundes ist) monolithisch und lokal betreibt. Folglich konzentriert sich die MANET-Forschung typischerweise auf die Entwicklung neuer Routing-Protokolle, die eine, den Bedingungen mobiler, drahtloser Netze *allgemein* (Fehlerträchtigkeit, Mobilität) und eine *spezifisch* dem jeweiligen Anwendungszweck (Unicast, Multicast, Robustheit, etc.), angepasste Kommunikation zur Verfügung zu stellen. Daraus resultieren oben genannte Voraussetzungen bezüglich der Kommunikation: Daten werden von Anwendungen beim Endgerät erstellt und über MANET-Routingprotokolle transparent (d. h. anwendungsunabhängig) an andere Endgeräte übermittelt.

Kollaborative und kollektive Sensornetze

In drahtlosen Sensornetzen stellt sich die Situation grundsätzlich anders dar. Während für die Kommunikation auf dem drahtlosen Medium dieselben Eigenschaften (und damit dieselben Herausforderungen) gelten, bestehen bezüglich der Kommunikations- und Anwendungsparadigmen gänzlich andere Voraussetzungen. Da intelligente Sensorknoten aufgrund ihrer Größe und ihres Anwendungsbereichs prinzipbedingt wesentlich weniger leistungsfähig sind als MANET-Knoten und zudem meist keine Benutzerschnittstelle im herkömmlichen Sinn anbieten, kann der stark Endgerät-bezogene Ansatz weder bei der Kommunikation noch bei der Anwendung aufrechterhalten werden. Die Betrachtung von typischen Anwendungen für drahtlose Sensornetze zeigt, dass eine dedizierte *Anwendung* auf einem benutzerinteragierenden Knoten nicht existiert. Die eigentliche Anwendung läuft vielmehr verteilt im gesamten Sensornetz ab, wobei die einzelnen Sensorknoten entweder *kollaborativ* und verteilt eine Anwendung ausführen, ohne dass der Benutzer (beziehungsweise ein ausgezeichneter, starker Steuerknoten) Ergebnisse beziehungsweise Daten zwingend erhält, oder die einzelnen Sensorknoten *kollektiv* ein Ergebnis erstellen und dieses dem Benutzer übermitteln. Abb. 2.1 zeigt modellhaft die Kommunikationsbeziehungen beider Paradigmen. Beide Paradigmen werden im Folgenden diskutiert.

Letztgenanntem Ansatz des *kollektiven* Sensornetzes widmen sich Forschungsarbeiten, die das Sensornetz als große Datenbank betrachten. Die Sensorknoten dienen als Datenersteller, die ein Phänomen beobachten beziehungsweise messen und die gewonnenen Daten einer Basisstation übermitteln. Die Anwendung teilt sich somit in zwei Teile: die Benutzerinteraktion, in der über die Basisstation dem Sensornetz eine Anfrage gestellt wird, und die Datensammlung, in der die Sensorknoten gemäß der Anfrage relevante Daten an die Basisstation senden beziehungsweise weiterleiten. Die linke Seite der Abb. 2.1 zeigt modellhaft diese Zweiteilung in Datenquelle (die Sensorknoten)

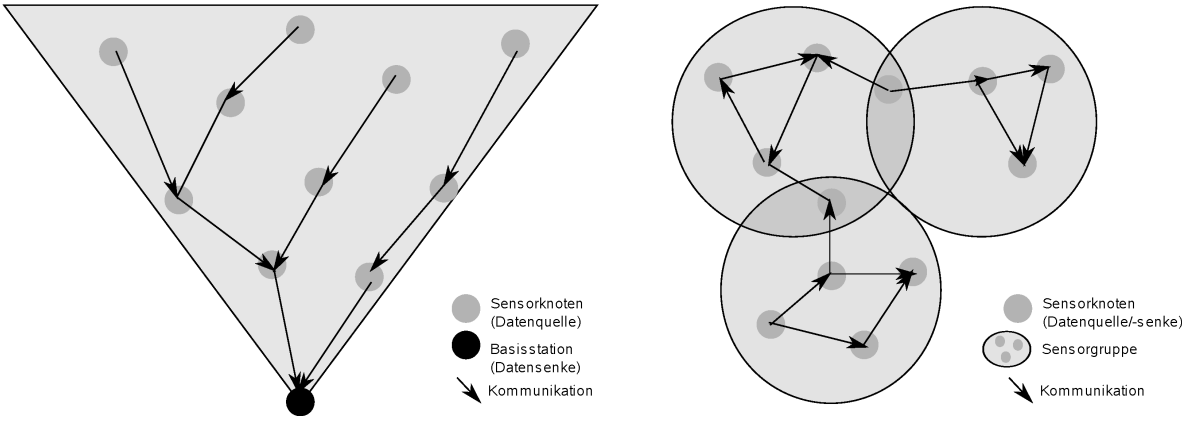

Abbildung 2.1: Prinzipielle Funktionsweise kollektiver (links) und kollaborativer Sensornetze (rechts)

und Datensenke (Basisstation). Die angefragten Daten werden immer in Richtung der anfragenden Basisstation weitergeleitet.

Die Herausforderung bei den kollektiven Sensornetzen liegt in einer geeigneten Darstellung der Benutzeranfrage (z. B. SQL-ähnlich), die deklarative Anfragen über Ort und Art der zu sammelnden Daten erlaubt, und in einer möglichst effizienten Weiterleitung der (ausschließlich relevanten) Daten an die Basisstation. Da hierbei im Gegensatz zu oben beschriebenen MANETs keine Endgerät-bezogene Adressierung benötigt wird, sondern ausschließlich über eine Datenbeschreibung adressiert wird, spricht man von *datenorientierten* Sensornetzen. In Netzen dieser Art werden die benötigten Daten über Schlüssel-Wert-Paare beschrieben, die das Netz (eigentlich die Sensorknoten) dazu nutzt, Daten zu erheben, diese gegebenenfalls zu verarbeiten (z. B. durch Aggregation) und zum anfragenden Knoten (d. h. in diesem Fall die Basisstation) weiterzuleiten. Eine Kommunikation *ausschließlich zwischen* Sensorknoten ist dabei nicht vorgesehen. Den Ansätzen zu *kollektiven*, d. h. *datenorientierten*, Sensornetzen gemeinsam ist eine Konzentration auf effiziente Weiterleitung großer Datenmengen. Oft wird dabei unter Ausnutzung redundanter Daten (durch Aggregation) oder periodisch wiederkehrenden Datenverkehrs (Optimierung der Wegewahl von den Quellen zur dedizierten Senke im Verlauf von wiederkehrenden Anfragen) die Kommunikation optimiert.

Datenorientierte Sensornetze

Bei *kollaborativen* Sensornetzen wird im Gegensatz zu den *kollektiven* Sensornetzen davon ausgegangen, dass Sensorknoten autonom Anwendungen beziehungsweise Anwendungsteile ausführen. Es ist nicht das alleinige Ziel, Daten beziehungsweise deren Aggregat zu einer vorher definierten Basisstation zu übermitteln, sondern auch die völlig autonome Ausführung von Anwendungen zu gestatten, die nie oder nur sporadisch

Daten „nach außen“ senden. Stellvertretend für solche Anwendungen stehen hierbei Regelungsaufgaben (z. B. Lichtregelung mit Lichtsensoren und -aktoren), Alarmmelder (z. B. Feuermeldung bei Überschreitung einer kritischen Temperatur bei einem oder einer Mindestanzahl von Temperatursensoren) oder Anwendungen, die variable Parameter für andere Anwendungen festlegen. Die rechte Seite der Abb. 2.1 zeigt modellhaft ein kollektives Sensornetz. Die Kreise gruppieren Sensorknoten, die einen Teilaspekt der Gesamtanwendung verwirklichen, beispielsweise drei separate Lichtregelkreise. Sensorknoten können auch bei mehreren Teilaspekten gleichzeitig mitwirken, beispielsweise Lichtaktoren, die für mehrere Lichtregelkreise verantwortlich sind (siehe dazu auch das intelligente Gewächshaus in Abschnitt 2.1.2).

Aufgrund der gewollten Beschränkungen des *datenorientierten* Ansatzes ist eine Umsetzung von kollaborativen Sensornetzen mit diesem nicht möglich. Da die vom Sensornetz generierten Daten ausschließlich aufgrund ihrer Meta-Beschreibung durch Schlüssel-Wert-Paare weitergeleitet werden, ist eine bidirektionale Kommunikation zwischen dedizierten Sensorknoten nicht oder nur unter besonderen Einschränkungen möglich. Denn nur der Empfänger ist ein ausgezeichneter Knoten (s. o. die Basisstation), während die Sender (d. h. die Quellen) über die Meta-Beschreibung *Knoten-unabhängig* gewählt werden. Dies wird insbesondere sichtbar, wenn „Sensornetze“ zu „Sensor-Aktor-Netzen“ verallgemeinert werden, d. h. nicht nur Sensoren zum Messen der Umgebung verwendet werden, sondern auch Aktoren zu deren Beeinflussung. Regelkreisläufe mit dedizierten Sensor- und Aktorknoten sind nur durch *netzweit eindeutige* Meta-Beschreibungen der Daten implementierbar, was dann einer Endgerät-bezogenen Adressierung entspräche. Allerdings sind spezielle Optimierungen wie Aggregation für datenorientierte Kommunikation in diesem Falle nutzlos und deren Zusatzaufwand somit überflüssig. Zu der erschwerten gezielten Kommunikation kommt hinzu, dass die eigentliche Anwendungslogik (bedingter Kontrollfluss, wiederholte Ausführung, etc.) beim datenorientierten Ansatz nicht vorgesehen ist, da der vorgegebene Anwendungszweck stets eine Anfrage an die „Datenbank Sensornetz“ ist.

2.2.1 Anforderungen an kollaborative Sensornetze

Um *kollaborative* Sensornetze im Sinne des vorangegangen Absatzes zu ermöglichen, ist eine alternative Sicht auf die Funktionsweise und die Anwendungsprogrammierung von Sensornetzen notwendig. Essentiell hierfür ist die Beschreibungsmöglichkeit von Anwendungslogik, die über eine reine Daten(bank)abfrage hinausgeht. Bei dieser Beschreibung der Anwendungslogik sollte auf die speziellen Eigenschaften von intelligenten Sensor-

knoten bezüglich ihrer Beschränkungen bei Rechenkapazität und Speicher Rücksicht genommen werden. Um Anwendungen in kollaborativen Sensornetzen zu ermöglichen und möglichst einschränkungsfrei formulieren zu können, sollte demnach die Umsetzung eines im obigen Sinne alternativen Ansatzes folgende Eigenschaften haben:

Anforderungen an kollaborative Sensornetze

1. Ermöglichung des lesenden/schreibenden Zugriffs auf Sensoren/Aktoren (Datenfluss),
2. Möglichkeit zur Formulierung von Anwendungslogik (Kontrollfluss),
3. Unterstützung einer verteilten Ausführung,
4. kleine atomare Einheiten.

Der erste Punkt entspricht den grundsätzlichen Anforderungen an ein allgemeines Sensor-Aktor-Netz: Sensoren sollen „lesend“ abgefragt, Aktoren dementsprechend „schreibend“ gesteuert werden können. Die zweite Eigenschaft betrifft die zentrale Anforderung an kollaborative Sensornetze: Durch die Möglichkeit zur Formulierung von Anwendungslogik soll eine neue Klasse von Anwendungen in Sensornetzen ermöglicht werden.

Während die ersten beiden Punkte funktionale Anforderungen betreffen, die der generellen Programmierbarkeit von (Sensor-)Netzen zuzuordnen sind, garantieren die beiden letztgenannten, nicht-funktionalen Anforderungen eine Umsetzbarkeit speziell in Sensornetzen: Sensorknoten sind im Gegensatz zu Knoten anderer Netze (z. B. MANETs) in der Regel wesentlich eingeschränkter bezüglich Speicher, Rechenkapazität und Energie. Die Unterstützung einer verteilten Ausführung auf mehreren Knoten ist deshalb für Sensornetze unabdingbar. Ohne eine Möglichkeit zur verteilten Ausführung einer Anwendung müsste diese (abgesehen vom Zugriff auf entfernte Sensoren und Aktoren) monolithisch auf genau einem Knoten ablaufen. Je nach Größe und Komplexität der Anwendung ist dies aber wegen der Speicher- beziehungsweise Rechenkapazitätsbeschränkungen nicht möglich (beziehungsweise nur auf, bezüglich Speicher und CPU, besonders ausgestatteten Knoten). Die Energieknappheit wirkt sich hingegen nicht nur auf den Knoten mit der monolithischen Anwendung aus, indem dieser sämtliche Energiekosten für Kommunikation und Rechenzeit trägt, sondern auch auf die ihn umgebenden Knoten, die als weiterleitende Knoten für Multi-Hop-Verbindungen zu Sensoren/Aktoren dienen.

Die letzte Forderung der möglichst kleinen atomaren Einheiten ist für Sensornetze von besonderer Wichtigkeit. Da Sensorknoten, wie oben beschrieben, knappe Ressourcen bezüglich Rechenkapazität und Speicher haben, sind möglichst kleine, atomare Einheiten essentiell. Atomare Einheiten sind hier als „auszuführender Code“ definiert, der zwin-

gendermaßen auf einem einzelnen Sensorknoten ausgeführt werden muss und nicht auf verschiedene Knoten verteilt werden kann. Dabei ist die „Größe“ der atomaren Einheiten auf sowohl den Speicherbedarf als auch den Rechenaufwand bezogen. Je kleiner in diesem Sinne die atomaren Einheiten sind, desto größer sind die Chancen, sie auf schwachen Sensorknoten speichern beziehungsweise ausführen zu können, und desto größer sind die Möglichkeiten der Anwendungsverteilung.

2.2.2 Dienstorientierung als Lösungsansatz für kollaborative Sensornetze

Dienstorientierte Sensornetze

Aus obigen Überlegungen folgt die Notwendigkeit eines neuen, vom *datenorientierten* Ansatz abweichenden, Paradigmas für Sensornetze. Besonders vielversprechend ist ein *dienstorientierter* Ansatz. Dienstorientierte Sensornetze eignen sich im Besonderen für oben erwähnte *kollektive* Sensornetze mit ihren speziellen Anforderungen. Dienste können in ihrer Funktionalität und Größe sehr feingranular definiert werden, so dass sowohl die funktionalen wie auch die nicht-funktionalen Anforderungen erfüllt werden.

Die Definition eines Dienstes folgt einem festgelegten Bauplan, der als Diensttyp bezeichnet wird. Ein Beispiel für einen Diensttyp wäre ein Bauplan, mit dem unter Angabe gewisser Attribute ein Dienst zum Auslesen eines physikalischen Sensors realisiert werden kann. Ein Diensttyp kann auch als *elementare Anweisung* einer Programmiersprache verstanden werden; die Gesamtheit der Diensttypen wären dann der *Befehlssatz*.

Zur Erfüllung der funktionalen Anforderungen müssen die zur Verfügung stehenden Diensttypen so festgelegt werden, dass mit diesen Daten- und Kontrollfluss einer Anwendung definiert werden können.

Zur Erfüllung der nicht-funktionalen Anforderungen muss die Anzahl der Diensttypen möglichst klein gehalten werden. Je größer die Zahl der Diensttypen ist, desto aufwendiger wird für den einzelnen Sensorknoten die Ausführung, da jeder Diensttyp nach der ihm eigenen Semantik interpretiert und ausgeführt werden muss. Wächst die Anzahl der zur Verfügung stehenden Diensttypen, steigt auch die Komplexität des für diesen größeren „Befehlssatz“ notwendigen Interpretierers. Damit die Ausführung aller Diensttypen auch ressourcenschwachen Sensorknoten möglich ist, sollte deswegen die Größe des „Befehlsatzes“ und damit die Anzahl der Diensttypen möglichst klein sein.

Gleichzeitig sollten die Diensttypen trotz ihrer geringen Zahl sehr wenig Informationsgehalt haben (d. h. geringen Speicheraufwand), deren Ausführung einfach sein (im

Sinne geringen Rechenaufwands) und gleichzeitig die Implementierung auch komplexer Anwendungen erlauben. Gerade letztgenannten Punkt unterstützt der dienstorientierte Ansatz durch die Möglichkeit zur Komposition verschiedener Dienste zu größeren Einheiten. Trotz Beschränktheit und Einfachheit der grundlegenden Dienste sind somit komplexe Anwendungen möglich.

Dienstorientierte Systeme im Allgemeinen werden nach [GZ06] folgendermaßen definiert:

> „Ein System kann von sich aus aktiv werden, und Aufträge an andere Systeme erteilen. Oder es bietet Dienste an, die von anderen durch einen Auftrag in Anspruch genommen werden können. [...] “

In Abschnitt 3 wird eine dienstorientierte Architektur, eine Dienst-Definitionssprache und ein Programmiermodell für dienstorientierte Sensornetze entwickelt, die die einzelnen Sensorknoten als solches System definieren. Architektur, Definitionssprache und Programmiermodell erlauben sowohl die Definition von Diensten als auch die Erteilung von Aufträgen im Sinne der obigen Definition.

2.3 Verwandte Forschungsarbeiten

Im Folgenden werden Forschungsarbeiten und -ergebnisse dargestellt, die in engem Zusammenhang mit dieser Arbeit stehen. Zum einen sind dies Programmierabstraktionen, die die Programmierung von Sensornetzen unterstützen, und damit vergleichbare Ziele verfolgen wie die in dieser Arbeit in Kapitel 3 entwickelte dienstorientierte Sprache und Architektur PanTalassa. Die Programmierabstraktionen werden in Abschnitt 2.3.1 dargestellt. Zum anderen werden in Abschnitt 2.3.2 relevante Forschungsarbeiten diskutiert, die die Kommunikation oder ihre anwendungsgesteuerte Beeinflussung zum Thema haben und damit vergleichbare Ziele verfolgen wie die in dieser Arbeit in Kapitel 4 entwickelte Programmierabstraktion zur modusbasierten Optimierung der Kommunikation.

2.3.1 Programmierung in Sensornetzen

Die große Vielfalt an Anwendungen führt zu unterschiedlichsten Anforderungen an die Programmierung von Sensornetzen. [RKM02], [HM06] und [MP11] fassen die Anforderungen zusammen und klassifizieren die bestehenden Programmierabstraktionen nach ihren zugrundeliegenden Eigenschaften. Im Folgenden werden relevante Forschungsarbeiten vorgestellt.

Nach [MP11] existieren hinsichtlich der Anwendungsaufgabe zwei Hauptmodelle für die Programmierung von Sensornetzen: *sense-only* und *sense-and-react*. Auf die gleiche Weise kategorisiert auch [MI05]: *database* und *active-sensor*. Die erste Kategorie entspricht dem in dieser Arbeit verwendeten Begriff der „kollektiven Sensornetze“. Sie dienen ausschließlich dem Erheben, Sammeln und Auslesen von Sensordaten. Die zweite Kategorie entspricht dem in dieser Arbeit verwendeten Begriff der „kollaborativen Sensornetze“. In ihnen erhalten Sensorknoten die Möglichkeit auf selbst oder von anderen generierte Daten zu reagieren.

Programmierabstraktionen für kollektive Sensornetze

Zu den Programmierabstraktionen, die kollektive Sensornetze unterstützen gehören u. a. Abstract regions, Cougar, DFuse, EnviroTrack, SINA und TinyDB.

Abstract Regions [WM04] basiert auf der Erkenntnis, dass in vielen Anwendungen eindeutig abgrenzbare Gruppen von Sensorknoten eine Aufgabe gemeinschaftlich erfüllten. Abstract Regions konzentriert sich dabei auf die Verarbeitung und Verteilung von Daten. Dazu stellt es eine Reihe von Kommunikationsprimitiven bereit, die in einem lokalen Bereich Methoden zur Adressierung, Datenaggregation und die gemeinsame Nutzung von Daten zur Verfügung stellen. Dies führt zu einer effizienteren Nutzung von Energie und Bandbreite, da eine lokalen Kommunikation und Datenverarbeitung gegenüber einer Datenübertragung bevorzugt wird.

Cougar [YG02] stellt das Sensornetz als virtuelle Datenbank dar, die über eine SQL-ähnliche Sprache abgefragt werden kann. Als Relationen stehen die Menge der Sensoren, deren Eigenschaften, und die zu messenden, physikalischen Sensorwerte zur Verfügung. Abfragen können einmalig oder auch inkrementell über eine lange Beobachtungsdauer erfolgen.

DFuse [KWA+03] erlaubt eine abstrakte Definition von Datenaggregationen. In einer eigenen Fusions-API können Aggregierungsfunktionen und Datenströme definiert werden, die auch Datensynchronisierung, Speicherung und Fehlerbehandlung unterstützt. DFuse übernimmt dann die automatische Verteilung der Aggregierungsfunktion im Sensornetz, wobei die Verteilung periodisch neu berechnet und ggf. verändert wird. Der Einsatz von DFuse ist auf hierarchische Aggregierungsfunktionen beschränkt.

EnviroTrack [ABC+04] wurde für die Implementierung von Ortungs- und Verfolgungsaufgaben entwickelt. Bewegliche Beobachtungsziele wie z. B. Tiere, Personen oder Fahrzeuge werden anhand charakteristischer physikalischer Eigenschaften definiert. Anhand dieser Charakteristik kann die Bewegung des Beobachtungsziels innerhalb des Sensornetzes beobachtet werden.

SINA [SSJ01] basiert auf einer relationalen Datenbank. Jeder Sensorknoten unterhält dabei eine vollständige, individuelle Tabelle mit seinen aktuellen Sensorwerten oder mit einer Reihe von historischen Sensorwerten. Eine Anwendung kann knotenübergreifend oder knotenspeziell Daten mit einer SQL-ähnlichen Sprache abfragen.

TinyDB [MFHH03], [MFHH05] ist ebenfalls eine Datenbank-basierte Programmierabstraktion. Sie bietet eine relationale Tabelle „SENSORS“, deren Spalten den Sensortyp, den individuellen Bezeichner des Sensorknotens, die verbleibende Batteriekapazität und die Sensordaten enthalten. Über eine SQL-ähnliche Sprache können Daten einmalig oder wiederkehrend abgefragt werden.

Zusammenfassend sind Programmierabstraktionen für kollektive Sensornetze, inklusive der oben erwähnten, aufgrund ihres Aufbaus nur wenig geeignet für eine direkte Zusammenarbeit zwischen Sensorknoten. Zentrales Anliegen dieser Programmierabstraktionen ist die Reduktion und Aggregation massiv erhobener Sensordaten auf ihrem Weg zur Senke. Sie bieten generell keine Möglichkeit, beispielsweise den Kontrollfluss einer Anwendung zu definieren oder Aktoren anzusprechen.

Programmierabstraktionen für kollaborative Sensornetze

Zu den Programmierabstraktionen, die kollaborative Sensornetze unterstützen, gehören u. a. Maté, Impala, MagnetOS, Kairos und SensorWare.

Impala [LM03] bietet auf Betriebssystemebene Schnittstellen zum Empfangen und Installieren von Anwendungen und Anwendungsprotokollen. Dazu stellt Impala grundlegende Funktionen zur Verfügung, um Aktionen zeitgesteuert zu starten, Pakete asyn-

chron zu senden und abstandsabhängig an Nachbarknoten zu fluten. Der Fokus liegt hierbei auf der Verwaltung und Versionierung der empfangenen Anwendungen. Die Anwendungen selbst unterliegen keiner Beschränkung, d. h. sie sind in einer frei wählbaren Sprache implementiert. Impala ist in dieser Hinsicht mehr ein dem Betriebssystem zugehöriger Dienst als zur Programmierung verteilter Anwendungen geeignet.

Maté [LC02] basiert auf einer Assembler-ähnlichen Sprache, die in einer eigenen virtuellen Maschine auf jedem Sensorknoten ausgeführt wird. Programme werden in 24-Byte-große Pakete aufgeteilt, die an beliebige Sensorknoten übertragen und dort in einer virtuellen Maschine interpretiert werden können. Somit ist es sehr einfach, verteilte Anwendungen im Sensornetz zu verteilen und auszuführen. Die limitierten, assemblerorientierten sprachlichen Möglichkeiten erschweren allerdings die Implementierung komplexer Anwendungen und führen zu unübersichtlichem, schwer wartbarem Code.

MagnetOS [BBD+02] basiert ebenso wie Maté auf einer virtuellen Maschine. Im Gegensatz zu Maté wurde jedoch mit Java eine Hochsprache gewählt. Der Programmierer schreibt normale Javaprogramme, ohne Berücksichtigung der verteilten Ausführung. MagnetOS übernimmt die transparente und automatische Verteilung anhand der in Java erzeugten Objekte. Dabei entscheidet jeder Knoten autonom über die Platzierung der Objekte, wobei versucht wird, die Kommunikationspfade möglichst kurz zu halten. Generell bietet MagnetOS durch Java eine mächtige Programmiersprache, um auch komplexe Anwendungen umsetzen zu können, und verspricht durch das automatische Verschieben von Objekten eine gewisse Energieoptimierung auf Kommunikationsebene. Allerdings sind die Anforderungen an Sensorknoten sehr hoch, da sie eine JVM (java virtual machine) implementieren müssen. Zudem ist die Verteilung beschränkt auf Java-Objekte, die potentiell sehr viel Daten und Code enthalten. Somit sind die Möglichkeiten der Verteilung sehr eingeschränkt.

Kairos [GKGM05] bietet einen einheitlichen Zugriff auf fundamentale Funktionen eines Sensorknotens an. Über eine kleine Menge von Programmierprimitiven können Sensorknoten adressiert werden, Variablen auf bekannten Sensorknoten gelesen und geschrieben werden, und auf die aktuelle Liste der direkten (1-hop) Nachbarn zugegriffen werden. Kairos vereinfacht zwar durch einheitliche Schnittstellen den Zugriff auf fundamentale Funktionen. Höhersprachliche Konstrukte zur z. B. Steuerung des Kontrollflusses, Verteilung von Anwendungen, etc. fehlen jedoch.

SensorWare [BHS03], [BHSS07] erlaubt die Ausführung Tcl-basierter Skripte. Die einzelnen Skripte können von Knoten zu Knoten verschickt werden, wobei im Skript

festgelegt werden kann, wie viel Energie benötigt wird (andere Kriterien sind nicht möglich). Knoten können so die Annahme eines Skripts verweigern. Die Adressierung der Tcl-Skripte ist allerdings nicht Knoten-übergreifend, so dass eine Verschiebung auf einen anderen Knoten nur möglich ist, wenn das betroffene Skript nicht durch andere Skripte adressiert wird. Durch die umfassenden Möglichkeiten der Tcl-Skriptsprache können komplexe Anwendungen schnell implementiert werden. Aufgrund der hohen Anforderungen von Tcl bezüglich Energie und CPU ist jedoch laut [MP11] SensorWare nur für vergleichsweise mächtige „Sensorknoten“ wie PDAs sinnvoll.

Zusammenfassend sind Programmierabstraktionen für kollaborative Sensornetze weit weniger zahlreich als die für kollektive Sensornetze. Sie stützen sich entweder auf bereits vorhandene Sprachen (wie Java und Tcl) und überfordern damit die meisten Sensorknoten aufgrund ihres Speicher-, Energie- und Rechenzeitaufwands, oder sie setzen auf ein eher niedriges Abstraktionsniveau, was die Entwicklung von komplexen Anwendungen behindert oder sogar unmöglich macht.

2.3.2 Anwendungsgesteuerte Kommunikationsoptimierung in Sensornetzen

Generell sind Sensorknoten in drahtlosen Sensornetzen limitierte Geräte in Bezug auf Energie, Zuverlässigkeit und Kommunikation. Deswegen existieren für drahtlose Sensornetze zahlreiche auf diese Limitierungen angepasste Kommunikationsprotokolle. Insbesondere die Energieknappheit wird als eine der Hauptherausforderungen in Sensornetzen angesehen. Cluster-basierte Ansätze (wie LEACH [HCB00], TEEN [MA01]) bauen eine Topologie auf, um die Kommunikation über Cluster-Heads zu optimieren und Ressourcen zu sparen. Der Einsatz von netzinterner Aggregation (z. B. energiebewusst bei SPIN [KHB99] oder transparent bei Directed Diffusion [IGE+03]), die Energieoptimierung auf Linkebene (MFCA [YCLZ01]) oder Energieoptimierung über das gesamte Netz (IDEA [CWW10]) dient ebenfalls einer energiesparenden Kommunikation. Neben der Energieeffizienz finden auch schnelle, zeitkritische Kommunikation (z. B. SPEED [HSLA03] oder [YAEW04]) und Robustheit ([Sto04], [DCF+09]) Beachtung. Generell finden sich für Kommunikation in Sensornetzen sehr viele Protokolle, die nur ein bestimmtes, vordefiniertes Ziel optimieren. Dies betrifft hauptsächlich die energieeffiziente, die schnelle, d. h. latenzminimierende, und die robuste Kommunikation. Die oben genannten Protokolle passen sich den sich ändernden Gegebenheiten im Sensornetz an, sobald diese das Optimierungsziel beeinträchtigen (durch Änderung der MAC-Parameter [KHB99], [YCLZ01], [HSLA03]; dynamische Umgehung ausgefallener oder ungünstig

gewordener Knoten [CWW10], [DCF+09]; strukturelle Neuorganisation durch Neuwahl ausgezeichneter Knoten [HCB00], [MA01]). Allerdings bleibt das Optimierungsziel an sich stets dasselbe.

Weniger Beachtung findet das Problem der Änderung des Optimierungsziels. Das bereits bei den Programmierabstraktionen erwähnte Impala [LM03] bietet eine solche Änderung des Optimierungsziels an, indem auf ein anderes Kommunikationsprotokoll gewechselt wird. Dazu wird die Implementierung des neuen Kommunikationsprotokolls an alle Sensorknoten geschickt, installiert und gestartet. Während dies für langfristige Änderungen, wie z. B. neue Versionen eines Protokolls, geeignet ist, ist es für eine kurzfristige Änderung und etwaiges Zurückkehren auf das frühere Protokoll ungeeignet. Sollte beispielsweise von einem energiesparenden auf ein schnelles, d. h. latenzminimierendes Protokoll gewechselt werden, würde während der Verteilung des neuen Protokolls die Kommunikation nur sehr eingeschränkt und langsam funktionieren, was im Widerspruch zum eigentlich Ziel stünde.

Eine Anpassung der Kommunikation bieten auch adaptive Protokollarchitekturen an ([DOH07] und [LBY10]). Aufgrund sich ändernder Bedingungen, wie z. B. Knotenausfall, Funkstörungen, etc. werden Protokolle adaptiert. Allerdings sind die Änderungen stets lokal begrenzt. Ein netzweiter, alle Knoten betreffender Wechsel eines Kommunikationsprotokolls ist nicht vorgesehen.

Eine generische Methode, um das Verhalten eines drahtlosen Sensornetzes zu kontrollieren, bieten Protokolle zur konsistenten Rollenverteilung (siehe [FR05] und [PH09]). In diesen Protokollen können den Sensorknoten beliebige Rollen, wie z. B. Clusterhead, Aggregationsknoten, etc. zugeteilt werden. Somit könnten auf allen Sensorknoten verschiedene Kommunikationsprotokolle implementiert werden, zwischen denen dann über verschiedene Rollenzuteilung gewechselt werden könnte. Allerdings stehen dabei Abstimmungen zwischen Gruppen von Sensorknoten im Fokus, um beispielsweise zu verhindern dass eine Rolle mehrfach vergeben wird. Ein netzweiter, alle Knoten gleichermaßen betreffender Wechsel wird nicht angestrebt. Obwohl prinzipiell möglich, ist ein netzweiter Wechsel wegen der aufwendigen Abstimmungsvorgänge mit diesen Protokollen zeitlich und energetisch ineffizient.

3 PanTalassa: Programmierabstraktion für dienstorientierte Sensor-Aktor-Netze

In diesem Kapitel wird eine dienstorientierte Architektur sowie eine Dienst- und Daten-Definitionssprache als Programmierabstraktion für dienstorientierte Sensornetze vorgestellt. Architektur und Definitionssprache werden unter dem Namen PanTalassa[1](Pan steht für **P**rocess and **A**ggregate **N**amed data, Talassa steht für **Ta**sking **La**nguage for **S**ervice-oriented **S**ensor/**A**ctuator Networks) zusammengefasst. PanTalassa ermöglicht die Definition von Diensten, die Definition von Daten, und deren kombinierte Verwendung zur Programmierung von dienstorientierten, verteilten Anwendungen mit der Möglichkeit zur netzinternen Datenverarbeitung.

In den ersten beiden Abschnitten wird die Programmierabstraktion Talassa für dienstorientierte Sensornetze entwickelt. In Abschnitt 3.3 wird analog zum dienstorientierten Talassa die datenorientierte Programmierabstraktion Pan entwickelt. Im Folgeabschnitt 3.4 wird gezeigt, wie sich beide Programmierabstraktionen zu PanTalassa vereinen lassen. Nach einer Darstellung der Umsetzung von PanTalassa schließt das Kapitel mit einer ausführlichen Evaluation.

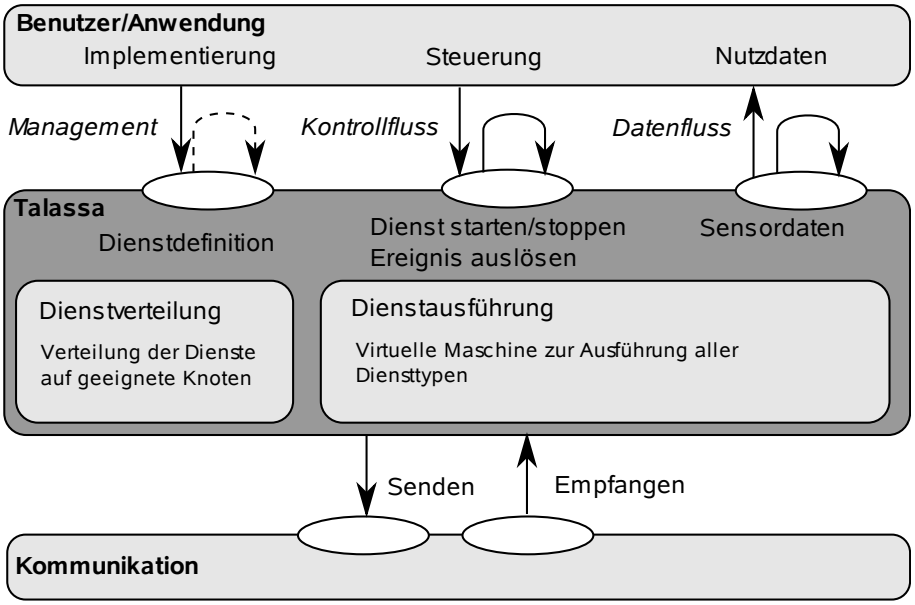

Abbildung 3.1: Architektur des dienstorientierten Ansatzes Talassa

3.1 Beschreibung dienstorientierter Sensornetze mit Talassa

Abb. 3.1 zeigt die Einordnung des Talassa-Systems in ein Schichtenmodell. Die Talassa-Schicht ist zwischen eine Anwendungsschicht und eine Kommunikationsschicht eingebettet. Die Kommunikationsschicht steht für die Schicht 3 des ISO/OSI-Schichtenmodells. Je nach Ausdehnung des Sensornetzes kann auch auf Wegewahl verzichtet werden und direkt die Schicht 2 verwendet werden. Für Talassa wird lediglich vorausgesetzt, dass jeder Sensorknoten mit jedem anderen kommunizieren kann. Ob dafür Zwischenknoten für eine Multi-Hop-Kommunikation verwendet werden oder die Kommunikation direkt erfolgt, ist für Talassa unerheblich. Für die Anwendungsimplementierung steht eine Management-Schnittstelle zur Verfügung, über die Dienste definiert (und gelöscht) werden können. Anwendungen bestehen in Talassa aus einer Komposition von vielen kleinen Diensten, einer sogenannten Dienstmenge.

[1]Der Name PanTalassa ist auch inspiriert durch das Urmeer (griech. *Panthalassa*), das den Urkontinent Pangaea vor drei Milliarden Jahren umflossen hat und als Ursprung allen Lebens und als Evolutionskeimzelle gilt.

Beispiel: [2] *Im Leitszenario des intelligenten Gewächshaus bilden die Dienste zur Feuchtigkeitsregelung eine Dienstmenge. Sie umfasst vier Dienste: einen Dienst zum Auslesen des Feuchtigkeitssensors, einen Dienst zum Steuern des Bewässerungsaktors, einen Dienst, der abhängig von der gemessenen Feuchtigkeit die Bewässerung startet, und einen Dienst, der für die regelmäßige Ausführung der vorhergehenden Dienste sorgt. Das Zusammenspiel dieser Dienste, ihre Steuerung und ihr Datenaustausch über die verschiedenen Schnittstellen werden im unten folgenden Beispiel erläutert.*

Um TALASSA-Anwendungen zu steuern, steht die Kontrollfluss-Schnittstelle zur Verfügung. Mit dieser können Dienste gestartet beziehungsweise gestoppt und Ereignisse ausgelöst werden. Auch Dienste können über diese Steuerung andere Dienste starten, stoppen oder Ereignisse auslösen, und damit den Kontrollfluss bestimmen. Sollte der Benutzer (Nutz-)Daten erwarten, erhält er diese über die Datenfluss-Schnittstelle.

Management-Schnittstelle

In der TALASSA-Schicht nimmt ein Modul die Dienste entgegen und sendet diese an die für den jeweiligen Dienst verantwortlichen Knoten. Während die Annahme von Diensten (über die Managementschnittstelle) naturgemäß jeder TALASSA -Sensorknoten beherrschen muss, ist die Verteilung (das Versenden an andere Knoten) nicht zwingendermaßen die Aufgabe *jedes* Sensorknotens. Diese Aufgabe kann auch nur von speziell geeigneten, im Extremfall nur von *einem* (Management-)Knoten übernommen werden. Nur für den Fall, dass Dienste auch autonom innerhalb des Sensornetzes verschoben werden können sollen, muss jeder Knoten Dienste sowohl annehmen als auch versenden können (dargestellt durch den gestrichelten Pfeil der Management-Schnittstelle). Dieses Verschieben von Diensten wird später in Abschnitt 3.6.2 zur Optimierung von Sensoranwendungen genutzt und evaluiert.

Kontrollfluss-Schnittstelle

Die eigentliche Ausführung der Dienste übernimmt eine virtuelle Maschine, die auf jedem Sensorknoten implementiert sein muss. Diese nimmt über die Kontrollfluss-Schnittstelle den Start/Stopp von Diensten beziehungsweise das Auslösen von Ereignissen an. Diese Schnittstelle wird nicht nur vom Benutzer oder der Anwendung verwendet, sondern wird auch intern von den TALASSA-Diensten genutzt, um den Kontrollfluss zu steuern. Abgesehen vom Kontrollfluss werden auch Sensordaten von Diensten generiert, die entweder über die Datenfluss-Schnittstelle an den Benutzer weitergegeben werden oder wiederum intern als Eingabe für andere Dienste dienen.

Datenfluss-Schnittstelle

[2] Im Folgenden werden immer wieder Beispiele für die dargestellten Architekturkomponenten, Mechanismen, etc. gegeben. Meist stellen diese Ausschnitte aus dem Leitszenario „Intelligentes Gewächshaus“ dar, das in Abschnitt 2.1.2 dargestellt wurde.

Alle drei Schnittstellen basieren auf einer Schnittstelle zur Kommunikationsschicht. Wenn zwei Dienste miteinander kommunizieren, muss die Kommunikationsschnittstelle deren Erreichbarkeit garantieren. Je nach Topologie ist eine Multi-Hop-Kommunikation (Schicht 3) notwendig oder eine direkte Kommunikation (Schicht 2) ausreichend.

Beispiel: *Die Dienste zur Feuchtigkeitsregelung aus dem vorigen Beispiel werden vom „Gärtner“ über die Managementschnittstelle an das Sensornetz gesendet, und die einzelnen Dienste werden auf den passenden Knoten gespeichert (der passende Knoten wird über einen Dienstparameter ermittelt, der in Abschnitt 3.2.1 eingeführt wird). Über die Kontrollfluss-Schnittstelle startet der Gärtner dann einmalig denjenigen Dienst, der regelmäßig die anderen Feuchtigkeitsdienste ebenfalls über die Kontrollfluss-Schnittstelle startet. Der Dienst, der den Feuchtigkeitssensor ausliest, sendet den Feuchtigkeitswert über die Datenfluss-Schnittstelle an den Dienst, der abhängig von diesem Wert über die Kontrollfluss-Schnittstelle den Bewässerungsdienst startet.*

3.2 Dienste in Talassa

Die Architektur in Abb. 3.1 beschreibt die abstrakte Form der Funktionsweise und der Schnittstellen eines dienstorientierten Sensornetzes basierend auf Talassa. Die konkrete Umsetzung der Architektur muss auf die Besonderheiten, insbesondere die Einschränkungen eines Sensornetzes beziehungsweise der Sensorknoten eingehen. Besonders die Implementierung der virtuellen Maschine und deren Ausführung darf die Fähigkeiten eines Sensorknotens nicht übersteigen. Die Anforderungen an die virtuelle Maschine sollten deswegen möglichst gering sein, so dass deren Implementierung eine möglichst geringe Komplexität (im Sinne des Rechenaufwandes) und möglichst geringen Speicherverbrauch aufweist. Die Komplexität der virtuellen Maschine hängt dabei unmittelbar mit den Diensten zusammen, die zu interpretieren und auszuführen sind. Dienste müssen auf den Sensorknoten sowohl gespeichert als auch interpretiert werden. Ihre Definition sollte deswegen „sparsam“ sein, ihre Anzahl „gering“. Nach dem Prinzip *Ockhams Skalpell*, das der Philosoph Johannes Clauberg wie folgt formuliert hat „*Entia non sunt multiplicanda praeter necessitatem*“, sollten „Entitäten nicht über das Notwendige vermehrt“ werden. „Sparsam“ soll in diesem Sinne ein Dienst sein, indem seine Ausführung keinen großen Rechenaufwand erfordert, und er durch wenige Attribute beschrieben werden kann. Somit wird garantiert, dass auch einfache Sensorknoten ohne große Speicher- und Rechenkapazität einen Dienst ausführen können. Die

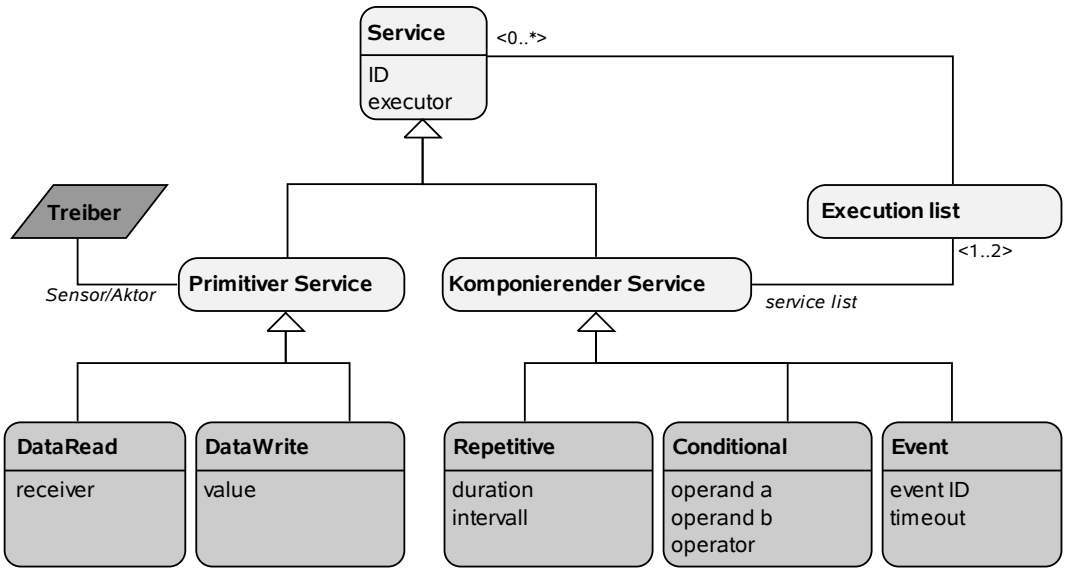

Abbildung 3.2: Grundlegender Aufbau eines Dienstes

„geringe“ Anzahl bezieht sich insbesondere auf die Anzahl der zur Verfügung stehenden Diensttypen.

Diensttypen sind eine Klassifizierung von Diensten. Jeder Diensttyp besitzt eine spezielle Funktionsweise und spezielle Attribute. Die Ausführung eines konkreten Dienstes hängt dann von seinem Diensttyp und der konkreten Attributbelegung ab. Diensttyp und Dienst im dienstorientierten Sensornetz stehen in der gleichen Beziehung wie Klasse und Objekt bei objektorientierten Programmiersprachen.

Im Folgenden wird versucht, wenige grundlegende Diensttypen zu finden, die trotzdem die gesamte Fülle aller frei nach Turing „intuitiv vorstellbarer“ Anwendungen zu implementieren erlauben, und die einzelnen Dienste in Umfang und Komplexität möglichst einfach zu halten. Der Umfang betrifft dabei die Anzahl der Attribute, die für die Definition eines Dienstes notwendig sind. Die Komplexität bezieht sich auf den zeitlichen und algorithmischen Aufwand, einen Dienst auszuführen.

Abb. 3.2 stellt den grundlegenden Aufbau eines Dienstes dar. TALASSA-Dienste werden durch Attribute (z. B. *serviceID*, *executor*, etc.) definiert. Dabei gibt es Attribute, die jeder Dienst definiert, und Attribute, die sich in Art und Anzahl nach dem jeweiligen Diensttyp unterscheiden. Die Diensttypen unterteilen sich in zwei Arten: *primitive* Diensttypen und *komponierende* Diensttypen.

Primitive Dienste

Primitive Dienste (siehe primitiver Service in Abb. 3.2) sind verantwortlich für die Ein- und Ausgabe von Daten. Wie in Abb. 3.2 erkennbar, nutzen sie hierfür einen *Treiber*, der außerhalb der virtuellen Maschine von Talassa (im Weiteren kurz TalassaVM genannt) implementiert ist. Dieser *Treiber* ist im Wesentlichen die hardwareabhängige Implementierung zum Auslesen von Datenproduzenten (physikalische Sensoren oder beliebige andere Datenproduzenten) und zum Schreiben von Datenkonsumenten (physikalische Aktoren oder beliebige andere Datenkonsumenten). Über das Attribut *Treiber* wird die jeweils auf dem Knoten zur Verfügung stehende Treiberimplementierung ausgewählt, die die Daten zurückgibt oder aufnimmt. Der Ein- und Ausgabe von Daten entsprechend gibt es zwei primitive Diensttypen *DataRead* und *DataWrite*.

Beispiel: *Im Leitszenario des intelligenten Gewächshauses sind die Dienste, die den Feuchtigkeitssensor beziehungsweise den Lichtsensor auslesen, vom Typ* DataRead*. Die Dienste, die die Wasserversorgung beziehungsweise die Pflanzenlampe steuern, sind vom Typ* DataWrite*.*

Komponierende Dienste

Komponierende Dienste (siehe komponierender Service in Abb. 3.2) sind verantwortlich für den Kontrollfluss, indem sie andere Dienste steuern. Wie in Abb. 3.2 erkennbar, nutzen sie dazu eine beziehungsweise zwei Ausführungslisten (siehe *execution list*), die beliebig viele Dienste beliebiger Art (primitive und komponierende Dienste) enthalten.

Ausführungsliste

Die in der Ausführungsliste enthaltenen Dienste werden abhängig von den Attributen des jeweiligen Dienstes kontrolliert. In der Auführungsliste können Dienste gestartet oder gestoppt werden. Zusätzlich gibt es in der Ausführungsliste die Möglichkeit, Ereignisse zu feuern, auf die im speziellen *Event*-Dienst (siehe nächster Absatz) reagiert wird. Im Falle des *Repetitive*-Dienstes wird die Ausführungsliste wiederholt ausgeführt, d. h. die enthaltenen Dienste werden bei jeder Ausführung der Liste erneut gestartet beziehungsweise gestoppt, und die enthaltenen Ereignisse werden erneut gefeuert.

Es existieren drei komponierende Diensttypen: *Repetitive*, *Conditional* und *Event*. Der Dienst *Repetitive* führt die Ausführungsliste in regelmäßigen Abständen wiederholt aus. Die Attribute *interval* und *duration* spezifizieren hierzu die Rahmenbedingungen der periodischen Ausführung. Der Dienst *Conditional* überprüft die mit den Attributen *operand a*, *operator* und *operand b* übergebene Bedingung und führt abhängig von dessen Ergebnis die *then-list* beziehungsweise die *else-list* aus. Der komponierende Dienst *Event* besitzt die zwei Attribute *eventID* und *timeout*. Er führt damit zwei Elemente ein, die für Sensornetze von enormer Wichtigkeit sind. Zum einen erlaubt er den Sensorknoten beziehungsweise dem Sensornetz, ohne Rechen- und Kommunikationsaufwand passiv auf spezielle Situationen zu warten und darauf zu reagieren (im Gegensatz zum

Service	
serviceID	Kennung des Dienstes
description	Beschreibung des Dienstes
executor	Ausführender Knoten
spezielle Dienstparameter, abhängig vom Service type	

Tabelle 3.1: Allgemeine Attribute eines Dienstes

aktiven Überprüfen von Bedingungen, was mit einem Zusammenspiel der Dienst *Repetitive* und *Conditional* modellierbar wäre). Zum anderen führt das Attribut *timeout* den Zeitbegriff ein. Mit diesem ist es möglich, Zeitabstände definiert zu überbrücken, und entsprechende Aktionen damit zu verbinden. Dies spielt insbesondere dann eine Rolle, wenn Interaktion mit menschlichen Benutzern stattfindet.

Beispiel: *Im Leitszenario des intelligenten Gewächshauses werden Dienste vom Typ* Repetitive *genutzt, um die Dienste für die Feuchtigkeitsregelung und Helligkeitsregelung regelmäßig zu starten. Diese Dienste führen den Abgleich von Feuchtigkeit beziehungsweise Helligkeit einmalig aus. Ein Dienst vom Typ* Repetitive *sorgt somit dafür, dass die Regelungen fortwährend ausgeführt werden.*

Vom Typ Conditional *sind beispielsweise Dienste, die gemessene Feuchtigkeitswerte oder Helligkeitswerte mit Sollwerten vergleichen und bei Bedarf die passenden Aktoren steuern. Dienste vom Typ* Event *könnten eingesetzt werden, um bei allen Pflanzen eine Warnlampe leuchten zu lassen, falls die Helligkeitsregelung eine zu hohe Helligkeit feststellt. Die Dienstmenge für die Helligkeitsregelung muss in diesem Fall ein entsprechendes Ereignis auslösen, worauf die* Event*-Dienste jeder Pflanze reagieren.*

Der hier besprochene, allgemeine Aufbau des Talassa-Dienstaufbaus wird im Folgenden genau spezifiziert und ggf. erweitert.

3.2.1 Allgemeine Attribute eines Dienstes

In Tab. 3.1 sind die allgemeinen, für alle Dienste verpflichtenden Attribute aufgeführt. Die *serviceID* kennzeichnet einen Dienst eindeutig und netzweit. Die Eindeutigkeit der Kennung muss durch Maßnahmen gewährleistet sein, die außerhalb von Talassa liegen. In lokal begrenzten Sensornetzen kann dies durch administrative Vorgaben durchgesetzt werden, bei großen Sensornetzen bieten sich Verfahren an, wie sie beispielsweise die

IEEE zur Eindeutigkeit von MAC-Adressen vorgibt (Herstellerkennung bei Netzwerkadaptern, siehe [IEE11]) oder Java für die Eindeutigkeit von Klassennamen vorschlägt (Paketnamen in Java, siehe [Bal10]).

Die Kennung wird dazu verwendet, wenn auf den entsprechenden Dienst zugegriffen werden soll. Dies ist insbesondere bei Ausführungslisten der Fall, die in Abschnitt 3.2.4 noch erläutert werden.

Der individuelle Zugriff auf Dienste ist aber in Sensornetzen nicht immer sinnvoll, insbesondere dann nicht, wenn mehrere gleichwertige Dienste den gleichen Auftrag ausführen können. Dienste werden laut Definition in Abschnitt 2.2.2 „von anderen durch einen Auftrag in Anspruch genommen“. Die Definition der Gleichheit zweier Dienste und damit des ausgeführten Auftrags hängt von der speziellen Anwendung ab. Beispielsweise können alle Dienste, die auf ihrem Knoten den Temperatursensor auslesen, als gleich angesehen werden, wenn der Ort der Temperaturmessung irrelevant ist.

Um aus einer Menge von Diensten eine Auswahl treffen zu können, wird eine nichtindividuelle Adressierung benötigt, die mithilfe einer allgemeinen Dienstbeschreibung passende Dienste auswählt. Zu diesem Zweck dient das Attribut *description*. Die Belegung dieses Attributes ist abhängig von der zusammen mit Talassa verwendeten Methode zur *Dienstsuche* (siehe Abschnitt 3.5.3). Im einfachsten Fall könnten beispielsweise alle als gleich definierten Dienste diesselbe Belegung bei dem Attribut *description* haben.

Das Attribut *executor* zeigt an, auf welchem Knoten ein Dienst angeboten wird. Diese Information wird verwendet, um die Dienste bei der Definition auf die entsprechenden Knoten zu verteilen. Bei Diensten, die nicht auf physikalische Sensoren oder Aktoren zugreifen, ist der ausführende Sensorknoten nicht festgelegt. In Abschnitt 3.6.2 wird ein Verfahren erläutert, wie der ausführende Knoten automatisch und ohne Festlegung des Programmierers bestimmt werden kann.

Anschließend an die allgemeinen Attribute folgen in den weiteren Abschnitten die speziellen Attribute, die für jeden Diensttyp unterschiedlich sind.

3.2.2 Spezielle Attribute der primitiven Dienste

Tab. 3.2 und Tab. 3.3 zeigen die speziellen Attribute der primitiven Dienste *DataRead* und *DataWrite*. Sie erfüllen Punkt 1 der funktionalen Forderungen (siehe Abschnitt 2.2.1), indem sie Daten lesen beziehungsweise schreiben lassen. Zentrales

DataRead	
driver	Treiberkennung zum Auslesen des physikalischen Sensors
context	optional: weiteres Attribut für den Treiber
target	Expliziter Empfängerdienst des ausgelesenen Datums/ oder *initiator* für den aufrufenden Dienst als Empfänger

Tabelle 3.2: Spezielle Attribute des *DataRead*-Dienstes

DataWrite	
driver	Treiberkennung zur Steuerung des physikalischen Aktors
context	optional: weiteres Attribut für den Treiber
value	an den Treiber übergebener Wert zum Steuern

Tabelle 3.3: Spezielle Attribute des *DataWrite*-Dienstes

Merkmal dieser Dienste ist die Abhängigkeit von einem außerhalb von TALASSA implementierten Treiber. Dies bedeutet, dass für jeden physikalischen Sensor/Aktor eines Sensorknotens der Zugriff über Sensor-/Aktortreiber ermöglicht werden muss. Beim *DataRead* wird das vom Treiber erhaltene (Nutz-)Datum an denjenigen Dienst gesendet, der unter *target* angegeben ist. Der Dienst wird als *serviceID* angegeben, wenn ein anderer Dienst der Empfänger sein soll. Wenn der aufrufende Dienst das Nutzdatum selbst empfangen soll, wird als *target* „initiator“ angegeben. Der Dienst *DataWrite* nimmt den unter *value* angegeben Wert entgegen und steuert mit diesem den angegebenen Aktortreiber. Die Syntax und die Auswirkung des Wertes sind dabei abhängig von der Funktionsweise des entsprechenden Treibers.

Beispiel: *Ein Treiber für einen Aktor, der Texte auf einem Display darstellt, nimmt den Wert in der Dienstbeschreibung als Zeichenkette und stellt diesen ohne Berücksichtigung des vorigen Zustands dar. Um eine Klimaanlage zu steuern, kann der Wert in der Dienstbeschreibung dagegen relativ zum gegenwärtigen Zustand angegeben werden. Ein Treiber für die Klimaanlage nimmt dann die zustandsabhängigen Werte „Temperatur erhöhen“ beziehungsweise „Temperatur verringern“ entgegen. Der Übergabewert* value *ist dabei abhängig von der Implementierung. Statt der Zeichenketten „Temperatur erhöhen“ und „Temperatur verringern“ kann aus Effizienzgründen auch eine Ganzzahl „1“ beziehungsweise „-1“ als* value *verwendet werden.*

Auch im Leitszenario des intelligenten Gewächshauses kann der Treiber für Pflanzenlampe zustandslos mit einem konkreten Helligkeitswert gesteuert werden; oder zustandsabhängig mit der Angabe, ob die Helligkeit erhöht oder verringert werden soll.

Der Begriff des Sensor-/Aktortreibers legt eine Festlegung auf die Behandlung von hardware-physisch vorhandenen Sensoren/Aktoren nahe. Im weiteren Verlauf dieser Arbeit stellt dies den Normalfall dar und wird in der Evaluation auch so implementiert. Prinzipiell kann in Talassa jedoch der Treiber eine beliebige Implementierung darstellen, die sich nicht allein auf physikalische Sensoren/Aktoren beschränkt. So können sich hinter einem Treiber beliebige Methoden und Algorithmen verbergen.

Beispiel: *Ein Sensorknoten kann eine Treiberimplementierung anbieten, in der dieser Knoten als Basisstation eines* kollektiven *Sensornetzes dient und das Ergebnis einer netzweiten Sensorwert-Anfrage über einen* DataRead-*Dienst an* Talassa *weitergibt.*

Beispielsweise kann ein Sensorknoten über einen Treiber mit der Kennung „tinydb“ eine SQL-basierte Anfrage an das Sensornetz über TinyDB ([MFHH05], siehe auch Abschnitt 2.3.1) anbieten.

- serviceID = *„Durchschnittstemperatur“*
- target = *„initiator“*
- driver = *„tinydb“*
- context = *„SELECT AVG(temp) FROM sensors“*

Wird dieser Dienst auf einem Sensorknoten aufgerufen, wird der auf dem Knoten implementierte Treiber für TinyDB mit dem angegebenen SQL-Ausdruck gestartet. Transparent für Talassa *würde dann über TinyDB die Durchschnittstemperatur über alle Sensoren ermittelt und abschließend das Ergebnis an den aufrufenden Dienst („initiator“) übergeben.*

Die einzige Forderung für Treiber im Sinne von Talassa lautet, dass Dienste vom Typ *DataRead* Nutzdaten produzieren und Dienste vom Typ *DataWrite* Nutzdaten konsumieren. Um die Mächtigkeit dieser Treiberimplementierungen nicht einzuschränken, können diesen über das Attribut *context* weitere Argumente übergeben werden. Im obigen Beispiel ist die Nutzung von *context* in einem *DataRead*-Dienst für TinyDB erläutert. Im Abschnitt 3.2.5 wird *context* für die Zustandshaltung mit *DataWrite*-Diensten genutzt.

3.2.3 Spezielle Attribute der komponierenden Dienste

Tab. 3.4, 3.5 und 3.6 zeigen die speziellen Attribute der komponierenden Dienste *Repetitive*, *Conditional* und *Event*. Diese Dienste erfüllen Punkt 2 der funktionalen Anforderungen (siehe Abschnitt 2.2.1), indem sie den Kontrollfluss über ihre jeweiligen Attribute vollständig steuern lassen. Der Dienst *Repetitive* bietet die Möglichkeit, Ausführungslisten periodisch auszuführen, was durch seine speziellen Attribute *duration* und *interval* genauer spezifiziert wird. Die Einheit dieser Attribute ist nicht generell festgelegt, sollte jedoch bei einer Implementierung der TALASSAVM die Angabe kurzer Zeiten im Bereich von Sekunden als auch größerer Zeitspannen im Bereich von mehreren Stunden oder mehr erlauben. Mit dem Dienst *Repetitive* ist auch die einmalige Ausführung einer Liste möglich, wenn das Wiederholungsintervall *interval* gleich der Wiederholungsdauer *duration* ist.

Beispiel: *Im Leitszenario des intelligenten Gewächshauses startet ein* Repetitive-*Dienst regelmäßig die Kontrolle der Feuchtigkeit. In der Annahme, dass die Angaben in Sekunden sind würde die Attributbelegung folgendermaßen aussehen:*

- serviceID = *„Feuchtigkeitsregelung“*
- duration = *86400*
- interval = *10*
- execution list = *<starte Dienste zur Feuchtigkeitskontrolle>*

Der Dienst „Feuchtigkeitsregelung“ würde in diesem Falle einen Tag lang alle 10 Sekunden die Dienste zur Feuchtigkeitskontrolle starten.

Damit der „Gärtner“ obigen Dienst einmalig starten kann, könnte folgender Dienst definiert werden:

- serviceID = *„Starte_ Feuchtigkeitsregelung“*
- duration = *1*

Repetitive	
duration	Dauer der gesamten Wiederholung
interval	Intervall zwischen zwei Ausführungen
execution list	Liste mit auszuführenden Aktionen

Tabelle 3.4: Spezielle Attribute des *Repetitive*-Dienstes

Conditional	
operand a	erster Operand des Vergleichs
operand b	zweiter Operand des Vergleichs
operator	Vergleichsoperator
then-list	Liste mit im positiven Fall auszuführenden Aktionen
else-list	Liste mit im negativen Fall auszuführenden Aktionen

Tabelle 3.5: Spezielle Attribute des *Conditional*-Dienstes

Event	
eventID	Kennung des Ereignisses, worauf reagiert werden soll
timeout	Zeitdauer, wie lange auf das Ereignis gewartet werden soll
event-list	Liste mit im Ereignisfall auszuführenden Aktionen
timeout-list	Liste mit im Timeout-Fall auszuführenden Aktionen

Tabelle 3.6: Spezielle Attribute des *Event*-Dienstes

- interval = *1*
- execution list = *start „Feuchtigkeitsregelung“*

Der Dienst „Starte_ Feuchtigkeitsregelung“ startet in diesem Fall einmalig den oben erwähnten Dienst „Feuchtigkeitsregelung“.

Der Dienst *Conditional* ermöglicht bedingte Ausführungslisten durch eine Evaluation einer Bedingung, die durch die Attribute *operand a*, *operator* und *operand b* spezifiziert wird. Obwohl es möglich ist, sowohl eine positive als auch eine negative Ausführungsliste anzugeben, sollte bei *Conditional*-Diensten die nicht-funktionale Anforderung nach möglichst kleinen „Bausteinen“ (Punkt 4) bedacht werden. In vielen Fällen ist es möglich, statt zwei Ausführungslisten im selben *Conditional*-Dienst anzugeben, diesen in zwei Dienste aufzuteilen. Im ersten Dienst wird die positive Bedingung angegeben, im zweiten wird die Bedingung durch Wahl des komplementären Operators logisch negiert. Der erste Dienst erhält die positive Ausführungsliste (im Attribut *then-list*), der zweite die negative (ebenfalls im Attribut *then-list*). Dadurch wird zum einen die Größe des Dienstes verringert, zum anderen der Ausführungsort des Dienstes flexibilisiert (positiver und negativer Fall können dann ggf. von verschiedenen Knoten evaluiert werden). Ein unterschiedlicher Ausführungsort ist insbesondere von Nutzen, wenn die Ausführungsliste im positiven beziehungsweise negativen Fall unterschiedliche

Dienste auf jeweils anderen Knoten startet. In diesem Fall können die zwei *Conditional*-Dienste voneinander unabhängig auf kommunikationstopologisch günstigen Knoten platziert werden, so dass Kommunikation eingespart werden kann.

Beispiel: *Im Leitszenario des intelligenten Gewächshauses kann folgender* Conditional *-Dienst definiert werden, um die Feuchtigkeit zu regeln:*

- serviceID = *„Kontrolle_geringe_Feuchtigkeit“*
- operand a = *<Dienst zur Messung der Feuchtigkeit>*
- operator = *„≤“*
- operand b = *<Gewünschter Feuchtigkeitswert>*
- then-list = *<Dienst, der die Wasserzufuhr startet>*

Wenn der Dienst gestartet wird, vergleicht dieser die aktuelle und gewünschte Feuchtigkeit und startet ggf. die Wasserzufuhr. Ein weiterer Dienst mit dem inversen Operator „>“ muss dann die Wasserzufuhr wieder stoppen. Alternativ kann auch die else-list *des obigen Dienstes dazu genutzt werden.*

Der Dienst *Event* ermöglicht die Ausführung einer Liste (*event-list*) in Abhängigkeit eines Ereignisses. Ereignisse werden anhand einer eindeutigen Kennung (*eventID*) unterschieden. Wie bei den anderen eindeutigen Kennungen muss auch die Eindeutigkeit der Ereignis-Kennung durch Maßnahmen gewährleistet werden, die außerhalb von TALASSA liegen. Ereignisse können sowohl von außerhalb (über die Benutzerschnittstelle) als auch TALASSA-intern initiiert werden (über Ausführungslisten, siehe 3.2.4). Sollte nach einer definierten Wartezeit (*timeout*) das Ereignis nicht eintreten, wird eine alternative Liste ausgeführt (*timeout-list*).

Beispiel: *Beim intelligenten Gewächshaus könnte ein* Event*-Dienst verwendet werden, um eine Warnlampe leuchten zu lassen, wenn die Helligkeitsregelung den zulässigen Bereich nicht mehr herstellen kann. Die Helligkeitsregelung soll in diesem Fall das Ereignis „Helligkeit_zu_hoch“ feuern.*

- serviceID = *„Warnung“*
- eventID = *„Helligkeit_zu_hoch“*
- timeout = *86400*
- event-list = *<Dienst, der Warnlampe einschaltet>*
- timeout-list = *start „Warnung“*

execution list	
start *serviceID*	startet den Dienst mit der Kennung *serviceID*
stop *serviceID*	stoppt den Dienst mit der Kennung *serviceID*
fire *eventID*	löst das Ereignis mit der Kennung *eventID* aus
<wait_n>	setzt Synchronisationspunkt (siehe Tab. 3.8 und nächsten Abschnitt über Synchronisation)

Tabelle 3.7: Mögliche Aktionen in Ausführungslisten

Nach dem Start wartet dieser Dienst einen Tag lang auf das genannte Ereignis. Trifft dieses ein, wird die Warnlampe eingeschaltet und der Dienst wird beendet. Sollte das Ereignis nicht eintreten, startet in diesem Fall der Dienst sich selbst erneut.

3.2.4 Ausführungsliste

Sämtliche drei komponierenden Diensttypen haben mindestens eine Ausführungsliste, wodurch sich beliebige Dienste zu größeren Einheiten komponieren lassen. Ausführungslisten dienen zur Steuerung anderer Dienste und zum Auslösen von Ereignissen. Tab. 3.7 gibt die möglichen Aktionen innerhalb einer Ausführungsliste an. Diese können in beliebiger Reihenfolge und in beliebiger Anzahl in einer Ausführungsliste erscheinen. Die verschiedenen Aktionen werden nebenläufig ausgeführt, was eine parallele Ausführung der gestarteten (oder durch Ereignis ausgelösten) Dienste bedeutet. Durch diese inhärente Parallelität der Ausführungsliste wird die Leistungsfähigkeit des Sensornetzes durch parallele Ausführung möglichst vieler Dienste besser genutzt.

Beispiel: *Im Leitszenario des intelligenten Gewächshauses kann beispielsweise ein* Repetitive-*Dienst mit folgender Ausführungsliste verwendet werden:*

- serviceID = *„Kontrolle“*
- execution list =
  - *start „Feuchtigkeitskontrolle“*
  - *start „Helligkeitskontrolle“*

Die Dienste „Feuchtigkeitskontrolle“ und „Helligkeitskontrolle“ werden regelmäßig gestartet und laufen parallel ab.

Synchronisation mit <wait_n>	
<wait_0>	wartet auf den erfolgreichen Start der vorhergehenden Aktionen
<wait_1>	wartet auf die erfolgreiche Beendigung der vorhergehenden Aktionen
<wait_2>	wartet auf die erfolgreiche Beendigung der vorhergehenden Aktionen und ihrer Folgeaktionen
<wait_3>	wartet auf die erfolgreiche Beendigung der vorhergehenden Aktionen, ihrer Folgeaktionen und deren Folgeaktionen
...	...
<wait_*>	wartet auf die erfolgreiche Beendigung der vorhergehenden Aktionen und aller (transitiven) Folgeaktionen

Tabelle 3.8: Synchronisationspunkte in Ausführungslisten

Synchronisation durch synchronisierte Ausführungslisten

In den meisten Sensornetzanwendungen ist eine parallele Ausführung aller Dienste möglich, im Sinne der Ressourcenausnutzung und Effizienz sogar erwünscht beziehungsweise notwendig. Aus diesem Grund ist die parallele Ausführung der Dienste in TALASSA der Normalfall und erfordert in den Ausführungslisten keine explizite Angabe. Allerdings gibt es Situationen, in der die Reihenfolge verschiedener Anwendungsteile eine wichtige Rolle spielt. Dies ist beispielsweise dann der Fall, wenn das Sensornetz mit einem Benutzer über ein Display interagiert oder wenn ein Regelkreis zwischen Aktor und Sensor modelliert wird, wobei der lesende Zugriff auf den Sensor erst *nach* dem Steuern des Aktors geschehen soll (z. B. soll die Helligkeit erst *nach* dem Einschalten der Lampe gemessen werden). Allgemein ist eine synchronisierte Ausführung immer dann notwendig, wenn ein Anwendungsteil von einem Zustand abhängt, wobei der Zustand sich auf Folgendes beziehen kann:

1. Umweltzustand: Der physikalische Zustand der Umwelt kann sich spontan (ohne Einfluss des Sensornetzes) ändern oder über Aktoren aktiv verändert werden. Der geänderte Zustand kann von einem Benutzer erkannt (ohne aktive Messung des Sensornetzes, z. B. Ausgabe eines Displays) oder über Sensoren erfasst (Messung von Umwelteigenschaften, z. B. Temperatur) werden.

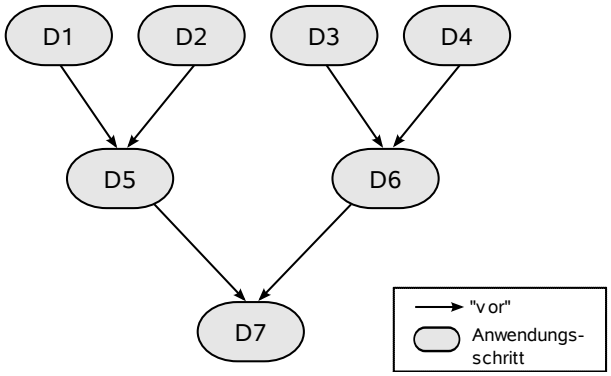

Abbildung 3.3: Verschachtelte Synchronisation

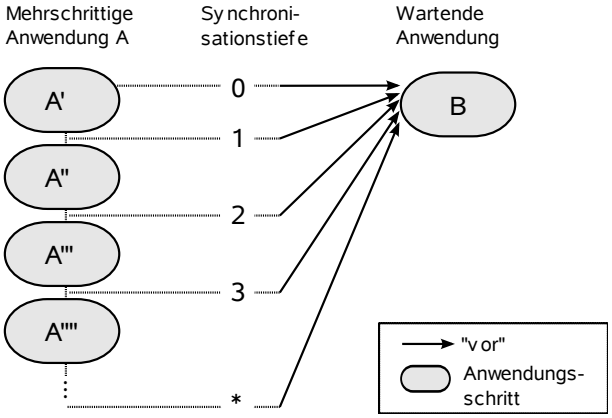

Abbildung 3.4: Tiefenabhängige Synchronisation

2. Speicher-/Variablenzustand: Der interne Zustand des Sensornetzes kann durch Einflussnahme über virtuelle (siehe Abschnitt 3.2.5) Aktoren verändert werden. Dieser wird durch Variablen (gespeichert durch einzelne Sensorknoten) repräsentiert und kann durch virtuelle Sensoren erfasst werden.
3. Anwendungszustand: Anwendungen beziehungsweise Anwendungsteile erreichen Zustände, die extern sichtbar werden (z. B. Displayausgabe wurde gesetzt) oder intern existieren (z. B. Sensormessung begonnen oder Sensormessung beendet).

Die Synchronisation ist demnach in gewissen Situationen unabdingbar für den korrekten Ablauf von Anwendungen. Synchronisation bedeutet jedoch nicht lediglich Sequenzialisierung aller Dienste. Besonders wenn Synchronisation den ansonsten parallelen Ablauf nicht unnötigerweise einschränken soll, müssen bei synchronisierten Anwendungen einige Aspekte zusätzlich betrachtet werden. Besondere Beachtung verdienen hierbei die beiden Aspekte:

1. Synchronisationstiefe und
2. Verschachtelung.

In Abb. 3.3 ist eine verschachtelte Synchronisation graphisch veranschaulicht. In diesem Falle erfolgt Anwendungsschritt D5, nachdem D1 und D2 erfolgt sind; Anwendungsschritt D6 analog, nachdem D3 und D4 erfolgt sind; D7 wiederum wartet auf die Ausführung von D5 und D6. Beim Ablauf können demnach D1 bis D4 parallel gestartet und ausgeführt werden. Sobald D1 und D2 (beziehungsweise D3 und D4) ausgeführt worden sind, kann mit der Ausführung von D5 (beziehungsweise D6) begonnen werden. Diese Art der Synchronisation kann z. B. bei der Aggregation von Daten beobachtet werden, wo die Vater-Aggregationsknoten unabhängig von anderen parallel ablaufenden Aggregationen auf die Daten ihrer jeweiligen Kinder-Knoten warten. Ein anderer Aspekt der Synchronisierung, wo ebenfalls der parallele Ablauf so wenig wie möglich beeinträchtigt werden soll, ist die Wahl der Synchronisationstiefe bei mehrschrittigen Anwendungen. Abb. 3.4 zeigt eine aus mehreren Schritten bestehende Anwendung A, die mit B synchronisiert werden soll. Die Anwendung B sollte in diesem Fall nur solange warten, bis der Zustand erreicht ist, auf dem B aufbaut. Die einzelnen Schritte der Anwendung A könnten beispielsweise Daten messen (A'), Daten senden (A'') und Daten speichern (A''') sein. Der gewünschte Synchronisationspunkt mit B könnte dann direkt nach dem Start von A erreicht sein (entspricht Tiefe 0), nachdem die Daten gemessen wurden (entspricht Tiefe 1), nachdem die Daten versendet wurden (entspricht Tiefe 2) usw. oder nach der vollständigen Abarbeitung aller Schritte (entspricht Tiefe *). Die jeweils verbleibenden Schritte nach dem Synchronisationspunkt können dann wieder uneingeschränkt parallel zu Anwendung B ablaufen.

Die obige Diskussion zeigt die Notwendigkeit für sowohl synchronisierte Abläufe als auch, im Zusammenspiel mit Parallelität, verfeinerte Beschreibungsmöglichkeiten der Synchronisation. Es sollte somit auch bei Talassa-Diensten eine Möglichkeit geben, synchronisierte Abläufe zu beschreiben. Dabei sollten die in Abschnitt 2.2.1 diskutierten Forderungen an Sparsamkeit eingehalten werden. Insbesondere sollten keine neuen Diensttypen oder völlig neue Konstrukte eingeführt werden, die die Knoten für die Synchronisation „über Gebühr belasten“. Die Komposition parallel und synchronisiert auszuführender Dienste sollte zudem nahezu „gleichartig“ erfolgen, um etwaige Änderungen bei der Umstellung von paralleler zu synchronisierter Ausführung gering zu halten.

Die „gleichartige“ Beschreibung ist von Vorteil, da somit eine Anwendung zunächst ohne Betrachtung der zeitlichen Abhängigkeiten modelliert werden kann. Beispielsweise können Regelkreise zuerst ohne Synchronisation beschrieben werden. In einem späteren

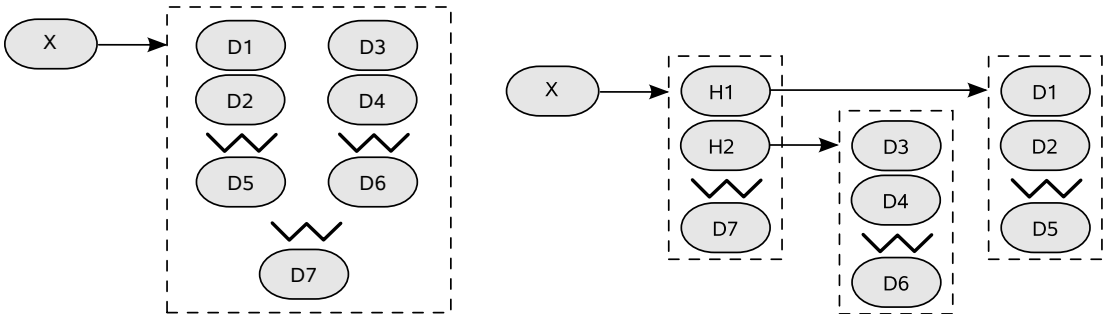

Abbildung 3.5: Umsetzungsvarianten der verschachtelten Synchronisation in Talassa: direkt (links) und mit Hilfsdiensten (rechts)

Schritt kann dann die parallele Ausführung auf eine synchronisierte umgestellt werden, ohne dass sich die grundlegende Modellierung ändert. Beispielsweise können dann Regelkreise synchronisiert werden, damit die betreffenden Sensoren des Regelkreises erst ausgelesen werden, nachdem die Aktoren aktiv geworden sind (z. B. Helligkeit messen erst nachdem die Lampenhelligkeit erhöht wurde).

Dienste werden in Talassa komponiert durch die Verwendung von Ausführungslisten. Durch sie können Dienste mit anderen Diensten (oder Aktionen) verbunden werden. Im Folgenden wird eine Möglichkeit zur Beschreibung der Synchronisation eingeführt, die sich vollständig auf die Ausführungslisten beschränkt und mit wenigen Ausdrucksmitteln alle Aspekte der Synchronisation berücksichtigt. Zentraler Punkt ist die Einführung von Synchronisationspunkten zwischen beliebigen Aktionen der Ausführungsliste. Ein Synchronisationspunkt bedeutet hierbei, dass sämtliche Aktionen *vor* diesem beendet sein müssen, bevor die weiteren Aktionen ausgeführt werden. Dies ermöglicht auf einfache Weise die Modellierung sequentieller Abläufe. Allerdings können verschachtelte Synchronisationsstrukturen nicht direkt auf diese Weise modelliert werden. Die naheliegende Möglichkeit, Synchronisationspunkte auch verschachtelt zuzulassen (siehe linke Seite Abb. 3.5), birgt allerdings für die dafür verantwortlichen Knoten ein hohes Maß an Komplexität. Da die Verschachtelung eine nahezu beliebige Rekursionstiefe erreichen kann, müssten die verantwortlichen Knoten bei der Abarbeitung einer solchen Ausführungsliste sehr viele Zustände gleichzeitig halten. Im Beispiel der Abb. 3.5 bedeutete dies das Warten auf die Erfüllung zweier Synchronisationspunkte gleichzeitig (zuerst Synchronisationspunkt (D1, D2) und (D3, D4); sobald einer dieser erreicht ist, kommt Synchronisationspunkt (D5, D6) hinzu). Abgesehen von der Problematik beliebig vieler Zwischenzustände ist auch die Interpretation *geschachtelter* Synchronisationspunkte wesentlich komplexer als die *ungeschachtelter*. Bei der geschachtelten Variante müssen Anfang und Ende der Synchronisation unterscheidbar markiert wer-

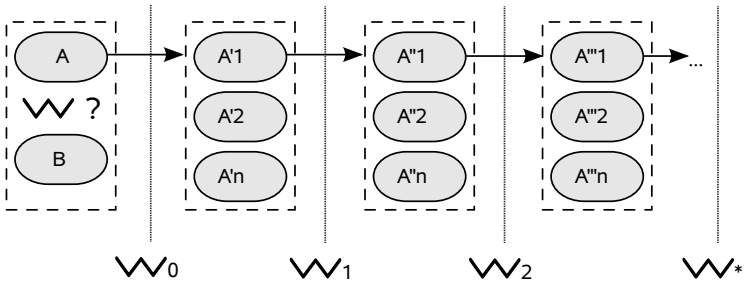

Abbildung 3.6: Umsetzung der variablen Synchronisationstiefe in TALASSA-Ausführungslisten

den, wohingegen die ungeschachtelte Variante ohne Anfangsmarkierungen (d. h. nur mit Synchronisations*punkten*) auskommt.

Durch den Aufbau der komponierenden Dienste besteht allerdings die Möglichkeit, geschachtelte Synchronisations*strukturen* mit ausführungsäquivalenten ungeschachtelten Synchronisations*punkten* zu modellieren. Abb. 3.5 zeigt auf der rechten Seite die ausführungsäquivalente Modellierung der linksseitigen Synchronisationsstruktur. Dazu wird für jede Synchronisationsgruppe (in diesem Fall „D5 wartet auf D1, D2“ und „D6 wartet auf D3, D4“) ein Hilfsdienst definiert (H1 und H2), der diese Gruppe synchronisiert ausführt (zuerst D1, D2, danach D5; und zuerst D3, D4, danach D6). Die Hilfsdienste werden dann wiederum in der hierarchisch höher gelegenen Synchronisationsgruppe stellvertretend für ihre Gruppen synchronisiert ausgeführt (in diesem Fall „D7 wartet auf H1, H2“).

Um auch solche Fälle in den Diensten abbilden zu können, können in Ausführungslisten Synchronisationspunkte gesetzt werden. In der Ausführungsliste werden diese durch $<wait\_n>$ gesetzt, wobei für n eine natürliche Zahl oder $*$ gesetzt werden kann (siehe Tab. 3.8). n spezifiziert die „Tiefe“ der Synchronisation. Dabei kann unterschieden werden, ob nur auf den erfolgreichen Start gewartet werden soll ($n = 0$) oder auf die komplette Ausführung ($n = 1$). Bei der Ausführung von Diensten, die wiederum eine Ausführungsliste beinhalten, kann auch auf deren erfolgreiche Ausführung ($n = 2$) gewartet werden. Die Synchronisation kann durch die Wahl von n beliebig tief erfolgen, mit $n = *$ wird auf *alle* (transitiv aufgerufenen) Dienste gewartet.

Beispiel: *Um eine Ampel in einer geplanten Reihenfolge aufleuchten zu lassen, kann eine Ausführungsliste wie folgt synchronisiert werden:*

- *start „Rot“*

- *<wait_ 1>*
- *start „Gelb“*
- *<wait_ 1>*
- *stop „Rot“*
- *stop „Gelb“*
- *start „Grün“*

Die Dienste „Rot“, „Gelb“, „Grün“ stehen hier für DataWrite-*Dienste, die die entsprechenden Farben einer Ampel leuchten lassen. Die Ausführungsliste wartet bei den entsprechenden Synchronisationspunkten bis die Ausführung der Dienste beendet ist (<wait_ 1>), und garantiert somit die korrekte Farbabfolge rot, rot-gelb, grün.*

Um die Durchschnittstemperatur zweier Räume zu ermitteln, die wiederum den Durchschnitt aus zwei Raumtemperatursensoren bilden, kann das Beispiel der verschachtelten Synchronisation mit den Diensten D1 bis D7 wie folgt umgesetzt werden:

- *Ausführungsliste X („Ermittle Durchschnitt beider Räume“)*
  - *start D5* („Ermittle Durchschnitt vom ersten Raum“)*
  - *start D6* („Ermittle Durchschnitt vom zweiten Raum“)*
  - *<wait_ 2>*
  - *start D7 („Berechne Durchschnitt beider Räume“)*
- *Ausführungsliste D5* („Ermittle Durchschnitt vom ersten Raum“)*
  - *start D1 („Temperatursensor 1 vom ersten Raum“)*
  - *start D2 („Temperatursensor 2 vom ersten Raum“)*
  - *<wait_ 1>*
  - *start D5 („Berechne Durchschnitt des ersten Raums“)*
- *Ausführungsliste D6* („Ermittle Durchschnitt vom ersten Raum“)*
  - *start D3 („Temperatursensor 1 vom zweiten Raum“)*
  - *start D4 („Temperatursensor 2 vom zweiten Raum“)*
  - *<wait_ 1>*
  - *start D6 („Berechne Durchschnitt des zweiten Raums“)*

Dienst X startet mit seiner Ausführungsliste die Ermittlung des Durchschnitts beider Räume und wartet bis zur Berechnung des Gesamtdurchschnitts mittels eines Synchronisationspunkts der Tiefe 2. In den beiden Räumen wird der Durchschnitt aus zwei

DataRead als virtueller Sensor	
driver	„Memory“
context	Name des Platzhalters, von dem geschrieben wird
target	Expliziter Empfängerdienst des ausgelesenen Datums/ oder *initiator* für den aufrufenden Dienst als Empfänger

Tabelle 3.9: Spezielle Attribute des *DataRead*-Dienstes als virtueller Sensor

DataWrite als virtueller Aktor	
driver	„Memory“
context	Name des Platzhalters, auf den geschrieben wird
modifier	Art der Operation (siehe Tab. 3.11)
value	Wert

Tabelle 3.10: Spezielle Attribute des *DataWrite*-Dienstes als virtueller Aktor

Temperatursensoren ermittelt, weswegen die Berechnung des Raumdurchschnitts auf deren Auslesen mit einem Synchronisationspunkt der Tiefe 1 warten muss.

3.2.5 Zustandshaltung über virtuelle Sensoren/Aktoren

Mit dem bisher vorgestellten TALASSA-System ist es möglich, Sensoren auszulesen, Aktoren zu steuern und den Kontrollfluss zu modellieren. Es besteht allerdings keine explizite Möglichkeit, Werte zu speichern, um später auf diese zurückzugreifen. Für diesen

modifier	Bedeutung
Set	Setzen des übergebenen Wertes
Add	Addieren des übergebenen Wertes zum vorhandenen
Subtract	Subtrahieren des übergebenen Wertes vom vorhandenen
Multiply	Multiplizieren des übergebenen Wertes mit vorhandenem
Divide	Dividieren des vorhanden Werts durch übergebenen
...	beliebige andere zweidimensionale Funktionen (MAX, MIN, AVG, etc.

Tabelle 3.11: Optionen für *modifier* beim virtuellen Aktor „Memory“

Zweck bietet sich das offene Konzept der primitiven Dienste an. Da diese mit beliebigen Treibern zusammenarbeiten und nicht auf physisch vorhandene Sensoren/Aktoren limitiert sind, kann mit einer speziellen Treiberimplementierung für „virtuelle Sensoren/Aktoren“ die Speicherung von Zuständen eingeführt werden, ohne den prinzipiellen Aufbau der Dienste zu ändern, beziehungsweise ohne neue Diensttypen einzuführen. Die Attribute der virtuellen Sensoren/Aktoren entsprechen denen des allgemeinen *DataRead-* beziehungsweise *DataWrite*-Dienstes. Nur greift die Treiberimplementierung auf den flüchtigen Speicher (RAM) des Sensorknotens zu, anstatt einen physikalischen Sensor auszulesen oder einen physikalischen Aktor zu steuern.

Diese „virtuellen Sensoren/Aktoren“ werden über die Treiberkennung *driver* „Memory“ angesprochen. Über das Attribut *context* wird ein Name vergeben, unter dem Werte gespeichert beziehungsweise gelesen werden. Um die Flexibilität noch zu erhöhen, kann das Attribut *modifier* noch zusätzlich Operationen auf den Daten anbieten (wie z. B. addieren, subtrahieren, etc.). Tab. 3.9 zeigt den Aufbau virtueller Sensoren, Tab. 3.9 entsprechend den Aufbau virtueller Aktoren (mit Beispielen für *modifier* in Tab. 3.11).

Beispiel: *Will man den Wert „1000“ auf dem Sensorknoten X zwischenspeichern, kann folgender* DataWrite-*Dienst genutzt werden:*

- serviceID = *„Speichere“*
- driver = *„Memory“*
- context = *„A“*
- modifier = *„Set“*
- value = *„1000“*

Soll ein Wert, der auf diesem Knoten unter „A“ gespeichert ist, wieder gelesen werden, kann folgender DataRead-*Dienst genutzt werden:*

- serviceID = *„Lese“*
- driver = *„Memory“*
- context = *„A“*
- target = *„initiator“*

3.2.6 Notation

Da das TALASSA-System keine Programmiersprache im herkömmlichen Sinne (imperativ, funktional oder logisch) ist, bietet sich keine bestehende Notation beziehungsweise Pseudonotation an. Dienste können auf verschiedene Weise notiert werden (graphisch, tabellarisch, funktional, über XML, etc.). Im Folgenden werden zwei Möglichkeiten vorgestellt, die in dieser Arbeit genutzt werden. Die funktionale Notation lehnt sich an die Schreibweise für mathematische Funktionen an. Die graphische Notation nimmt Anleihen aus den Darstellungsformen von UML.

Funktionale Notation

Die funktionale Notation richtet sich ausschließlich nach den allgemeinen und speziellen Attributen der Diensttypen. Es existieren keine vereinfachenden Schreibweisen, womit sich diese Notation insbesondere dann eignet, wenn es auf exakte Definitionen ohne Interpretationsspielraum ankommt. Der *Service type* wird zusammen mit der *serviceID* als Funktionsname verwendet und die verbleibenden Attribute gleich einer Abbildungsvorschrift notiert. Die folgende Dienstdefinition *Service type* („*serviceID* “) 3.1 gibt den allgemeinen Aufbau einer Dienstdefinition in funktionaler Notation wieder.

$$\textit{Service type}(\text{„}\textit{serviceID}\text{ “}) := \begin{cases} \textit{attribut 1} & = \text{„Attributinhalt“}, \\ \textit{attribut 2} & = \text{„Attributinhalt“}, \\ \vdots & \vdots \end{cases} \tag{3.1}$$

Für die speziellen Diensttypen wird die funktionale Notation dem Diensttyp entsprechend verwendet. Der spezielle Diensttyp (*DataRead*, *DataWrite*, *Conditional*, *Repetitive*, *Event*) wird mit den dazugehörigen Attributen angegeben.

Graphische Notation

Abb. 3.7 zeigt die graphische Notation von Diensten. Grundsätzlich werden Dienste mit einem abgerundeten Rechteck symbolisiert.

Die oberen beiden Notationen der Abbildung zeigen den Zugriff eines *DataWrite*-Dienstes beziehungsweise eines *DataRead*-Dienstes auf einen Treiber. Der gefederte Pfeil symbolisiert den Zugriff der primitiven Dienste auf einen Treiber. Wie in den vorangegangenen Abschnitten erläutert, ist der Treiber die Implementierung zum Steuern eines

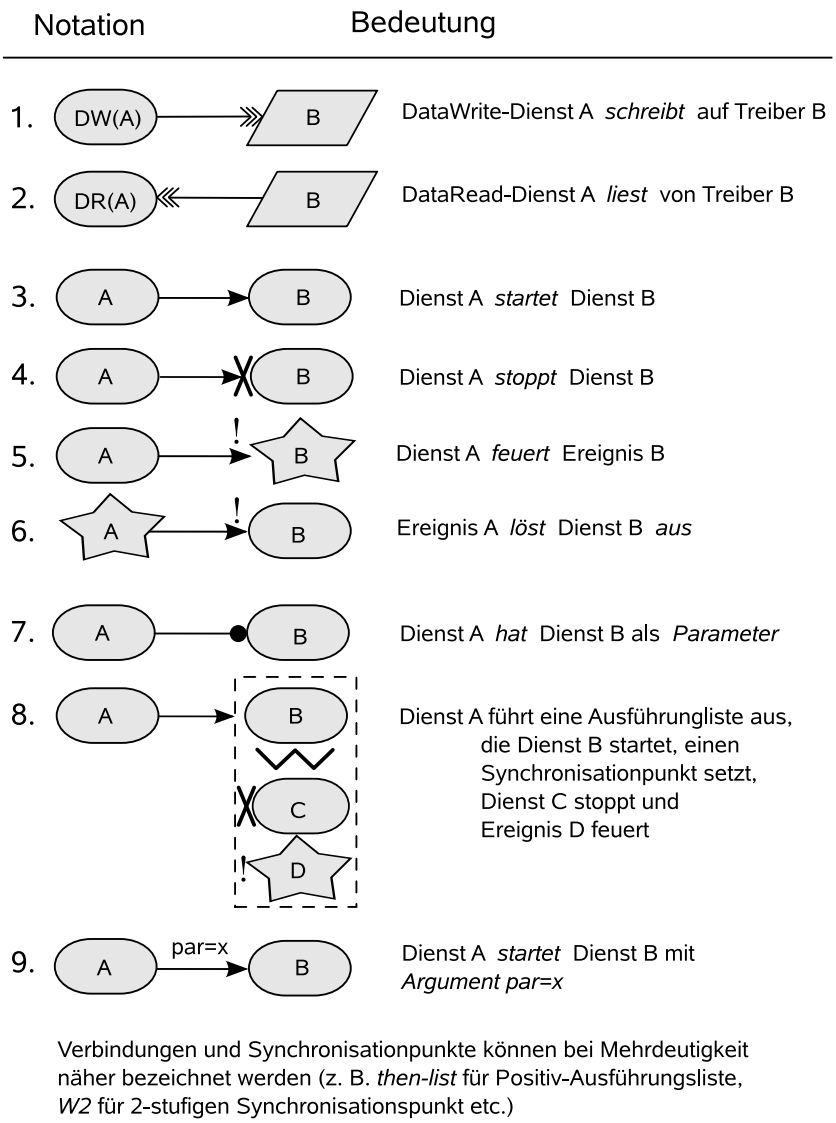

Abbildung 3.7: Graphische Notation von Talassa-Diensten und ihren Beziehungen

physikalischen Aktors beziehungsweise zum Auslesen eines physikalischen Sensors auf den Sensorknoten. Treiber werden durch eine Raute dargestellt.

Bei den komponierenden Diensten besteht die Möglichkeit, Dienste zu starten, zu stoppen, und Ereignisse auszulösen. Die graphische Repräsentation hierfür ist in den Zeilen 3 bis 5 der Abb. 3.7 dargestellt. Der einfache Pfeil steht für das Starten eines Dienstes, ein mit einem Kreuz abschließender Pfeil steht für das Stoppen, und ein mit einem Ausrufezeichen gekennzeichneter Pfeil steht für das Auslösen eines Ereignisses. Das Ereignis selbst wird mit einem Stern dargestellt. Das Auslösen eines *Event*-Diensts durch ein Ereignis wird ebenfalls mit einem durch ein Ausrufezeichen gekennzeichneten Pfeil dargestellt (Zeile 6 der Abb. 3.7).

Einige Dienste können *DataRead*-Dienste als variablen Parameter für ein Attribut besitzen. Dies betrifft den *DataWrite*-Dienst, der das Attribut *value* aus einem *DataRead*-Dienst bezieht. Bei den komponierenden Diensten können die Attribute *interval* und *duration* des *Repetitive*-Dienstes als *DataRead*-Dienst definiert werden, die Attribute *operand a* und *operand b* des *Conditional*-Dienstes, und das Attribut *timeout* beim *Event*-Dienst. Diese Attribute werden dadurch gekennzeichnet, dass sie durch eine mit einem Punkt abschließenden Linie an den entsprechenden *DataRead*-Dienst angebunden werden (Zeile 7 der Abb. 3.7).

Sämtliche komponierende Dienste können Ausführungslisten beinhalten. Als vereinfachende Darstellung können alle Dienste und Ereignisse einer Ausführungsliste in einem gestrichelten Rechteck zusammengefasst werden (Zeile 8 der Abb. 3.7). Innerhalb dieses Rechtecks können Synchronisationspunkte gesetzt und die geplante Aktion des Dienstes (starten beziehungsweise stoppen) entsprechend markiert werden. Sollte ein Dienst mehr als eine Ausführungsliste besitzen, wird der zur Ausführungsliste führende Pfeil mit dem Attributnamen der Ausführungsliste bezeichnet (d. h. für *Conditional*-Dienst *then-list*, *else-list*, und für *Event*-Dienst *event-list*, *timeout-list*).

Dienste mit Parameter können nicht nur über *DataRead*-Dienste variabel gestartet werden. Die Werte der Parameter eines Dienstes können auch direkt bei dessen Start übergeben werden. In diesem Falle wird der startende Pfeil mit der entsprechenden Parameterbelegung bezeichnet (Zeile 9 der Abb. 3.7).

Beispiel: *Für Beispiele der graphischen Notation von Diensten wird auf Abschnitt 3.6.4 verwiesen werden, wo das Leitszenario des intelligenten Gewächshauses modelliert und graphisch notiert ist.*

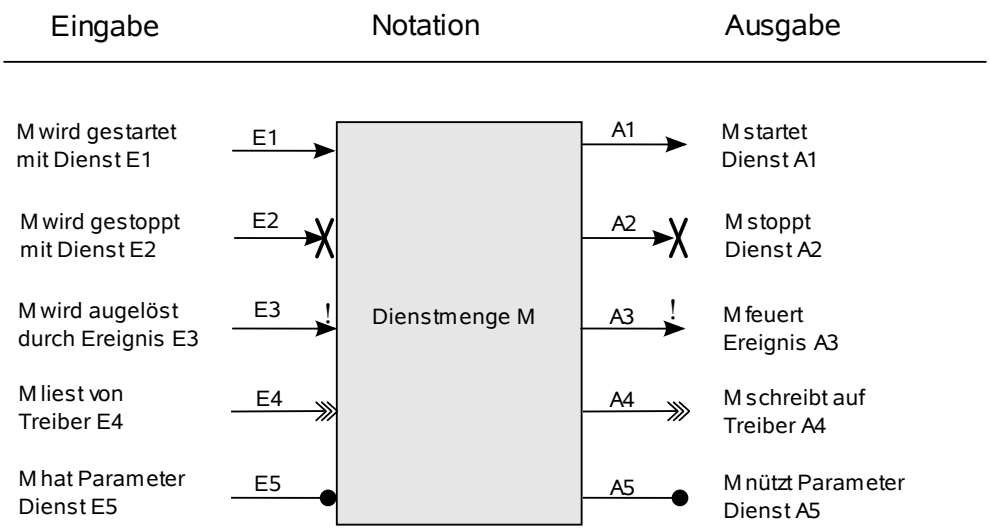

Abbildung 3.8: Graphische Notation komponierter Talassa-Dienste

In größeren Anwendungen können die Zahl der Dienste und ihre Zusammenhänge unübersichtlich groß werden. Es bietet sich deswegen an, Dienste gedanklich zu Dienstmengen zusammenzufassen. Eine Dienstmenge umfasst alle Dienste, die zur Ausführung einer definierten Aufgabe modelliert werden. Viele Dienste einer solchen Dienstmenge dienen ausschließlich zur Bearbeitung dieser Aufgabe. Sie werden durch (Dienstmengen-)eigene Dienste manipuliert beziehungsweise manipulieren andere (Dienstmengen-)eigene Dienste, indem sie diese starten, stoppen, etc. Nichtsdestoweniger ist es notwendig, dass die Dienstmenge von anderen Diensten oder Dienstmengen genutzt werden kann. Die Dienstmenge kann beispielsweise einen Dienst anbieten, der die ganze Dienstmenge startet oder der die Dienstmenge in seiner Ausführungsweise steuert. Ebenso wirkt die Dienstmenge nach außen, indem es andere Dienste oder Dienstmengen beeinflusst. Diejenigen Schnittstellen, die eine Manipulation von außen erlauben beziehungsweise nach außen wirken, müssen für eine Dienstmenge definiert werden.

Die Abb. 3.8 zeigt die graphische Notation für Dienstmengen. Die Dienstmenge wird durch ein Rechteck symbolisiert. Die Schnittstellen werden dadurch definiert, dass die relevanten Schnittstellen-Dienste, -Ereignisse und -Treiber durch die bei den Diensten eingeführten Pfeilnotationen nach außen geführt werden. Die Pfeile werden mit den Namen der Dienste, Ereignisse und Treiber gekennzeichnet.

Beispiel: *Für Beispiele der graphischen Notation von Dienstmengen wird auf Abschnitt 3.6.4 verwiesen.*

3.2.7 Mächtigkeit der Sprache

Im Folgenden soll die Mächtigkeit von TALASSA untersucht werden. Es wird eine Turingmaschine durch Dienste modelliert und dadurch der Nachweis der *Turing-Mächtigkeit* von TALASSA geführt. Die Dienste werden durch die oben eingeführte funktionale und graphische Notation definiert.

Gegeben Gegeben sei eine deterministische Turingmaschine $M = (Q, \Sigma, \sqcap, \Gamma, s, \delta, F)$, wobei

- Q: endliche Zustandsmenge,
- Σ: endliches Eingabealphabet,
- $\sqcup$: leeres Feld mit $\sqcup \notin \Sigma$,
- Γ: endliches Bandalphabet mit $\Sigma \cup \{\sqcup\} \subset \Gamma$,
- s: Startzustand,
- $\delta : (Q \backslash F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, N\}$: Überführungsfunktion,
- $F \subseteq Q$: Menge der Endzustände.

Ohne Beschränkung der Allgemeinheit wird davon ausgegangen, dass das Turingband nur einseitig unendlich ist und das leere Feld auf $\sqcup = 0$ definiert ist.

Behauptung TALASSA ist mindestens turingmächtig.

Beweis Zum Beweis der Turing-Mächtigkeit wird eine Menge von TALASSA-Diensten definiert, die die Turingmaschine M simuliert. Im Folgenden werden die einzelnen Teile (Zustände, Alphabet, etc.) vorerst einzeln modelliert, um sie im Anschluss zusammenzusetzen. Die Gesamtheit der konstruierten Dienste simuliert die Turingmaschine M.

Zustände

Für die Zustandshaltung wird das Konzept der virtuellen Sensoren/Aktoren benutzt. Unter dem *context* „Zustand“ wird der momentane Zustand gespeichert, bzw. gelesen. Zum Speichern der Zustände wird für jeden Zustand $q \in Q$ ein *DataWrite* mit der

serviceID =„SchreibeZustand_q“ definiert:

$$DataWrite(\text{„SchreibeZustand\_q“}) := \begin{cases} driver & = \text{„Memory“}, \\ context & = \text{„Zustand“}, \\ modifier & = Set, \\ value & = q \end{cases} \tag{3.2}$$

Zum Auslesen des Zustandes wird ein *DataRead* mit der *serviceID* =„LeseZustand“ definiert:

$$DataRead(\text{„LeseZustand“}) := \begin{cases} driver & = \text{„Memory“} \\ context & = \text{„Zustand“} \end{cases} \tag{3.3}$$

Lese-/Schreibkopf, Eingabealphabet, Bandalphabet

Das Band der Turingmaschine wird ebenfalls durch virtuelle Aktoren/Sensoren umgesetzt. Die Bandstellen werden mit fortlaufenden natürlichen Zahlen kodiert, die wiederum als *context* dienen. Um sich auf diesem „Band“ bewegen zu können, wird ein Lese-/Schreibkopf benötigt, der die aktuelle Bandstelle als natürliche Zahl enthält. Auch dieser Kopf wird über virtuelle Aktoren/Sensoren modelliert und dazu der Kontextname „Kopf“ benutzt. Bewegt, im Sinne der Turingmaschine, wird dieser Kopf mithilfe zweier *DataWrite*, die die Bandstelle um eins erhöhen bzw. erniedrigen, was einer Bewegung des Lese-/Schreibkopfes nach rechts bzw. links entspricht.

$$DataWrite(\text{„BewegeKopf\_}\{L, R\}\text{“}) := \begin{cases} driver & = \text{„Memory“}, \\ context & = \text{„Kopf“}, \\ modifier & = \{Add|Subtract\}, \\ value & = 1 \end{cases} \tag{3.4}$$

Zum Auslesen des Lese-/Schreibkopfes wird folgender *DataRead* definiert:

$$DataRead(\text{„LeseKopf“}) := \begin{cases} driver & = \text{„Memory“} \\ context & = \text{„Kopf“} \\ target & = initiator \end{cases} \tag{3.5}$$

Das eigentliche Band der Turingmaschine wird ebenfalls mit *DataWrite* beschrieben, bzw. mit *DataRead* ausgelesen. Beide nehmen in ihrem *context* auf die momentane Bandstelle bezug, der über *DataRead* „LeseKopf“ gelesen wird. Für jedes Alphabetzeichen $\gamma \in \Gamma$ wird ein *DataWrite* mit der *serviceID* =„SchreibeBand_γ definiert:

$$DataWrite(\text{„SchreibeBand\_}\gamma\text{“}) := \begin{cases} driver & = \text{„Memory“}, \\ context & = DataRead(\text{„LeseKopf“}), \\ modifier & = Set \\ value & = \gamma \end{cases} \tag{3.6}$$

Zum Auslesen der aktuellen Bandstellen wird folgender *DataRead* definiert:

$$DataRead(\text{„LeseBand“}) := \begin{cases} driver & = \text{„Memory“} \\ context & = DataRead(\text{„LeseKopf“}), \end{cases} \tag{3.7}$$

Überführungsfunktion

Die Überführungsfunktion hat als Eingabeparameter einen Zustand $q \in Q$ und das Alphabetzeichen $\gamma \in \Gamma$ der momentanen Bandstelle (auf die der Lese-/Schreibkopf zeigt), als Ausgabe einen Zustand $X' \in Q$, ein Alphabetzeichen $\gamma' \in \Gamma$, das auf die momentane Bandstelle geschrieben wird und eine Lese-/Schreibkopfbewegung $z \in \{L, R, N\}$. Da für eine Kopfbewegung $z = N$ (Stehenbleiben des Kopfes) kein gesonderter *DataWrite* definiert ist, soll zusätzlich gelten:

$$DataWrite(\text{„BewegeKopf\_}N\text{“}) := \emptyset \tag{3.8}$$

Jede definierte Stelle der Überführungsfunktion $\delta : (q, \gamma) \rightarrow (q', \gamma', z)$ wird nun in zwei verschachtelte *Conditional*-Dienste überführt. Zur Vereinfachung der Darstellung wurde der *start*-Befehl vor den einzelnen Einträgen der *execution list* weggelassen.

$$Conditional(\text{„}\delta(q,\gamma)\text{“}) := \begin{cases} operand\ a & = DataRead(\text{„LeseZustand“}), \\ operator & = \text{„==“}, \\ operand\ b & = q, \\ then\text{-}list^3 & = \{Conditional(\text{„}\delta(q,\gamma,\gamma')\text{“})\} \end{cases} \tag{3.9}$$

$$Conditional(\text{„}\delta(q,\gamma,\gamma')\text{“}) := \begin{cases} operand\ a & = DataRead(\text{„LeseBand“}), \\ operator & = \text{„==“}, \\ operand\ b & = \gamma, \\ then\text{-}list^3 & = \\ & \{DataWrite(\text{„SchreibeZustand\_}q'\text{“}), \\ & \ DataWrite(\text{„SchreibeBand\_}\gamma'\text{“}), \\ & \ DataWrite(\text{„BewegeKopf\_}z\text{“})\} \end{cases} \tag{3.10}$$

Initialisierung der Turingmaschine

Der Betrieb der Turingmaschine erfordert die Initialisierung des Bandes mit dem Eingabewort. Für jedes Alphabetzeichen $\sigma \in \Sigma$, das auf der Bandstelle $n \in \mathbb{N}$ gesetzt werden

[3] Die hier angegebenen Dienste werden gestartet.

soll, wird folgender *DataWrite* definiert:

$$DataWrite(\text{„SchreibeBand\_}n\text{“}) := \begin{cases} driver & = \text{„Memory“}, \\ context & = \text{„n“}, \\ modifier & = Set \\ value & = \sigma \end{cases} \tag{3.11}$$

Der „Rest“ des Turingbandes muss jeweils „leeren Feldern“ vorinitialisiert sein. Hier wird davon ausgegangen, dass nicht zugewiesene Kontextnamen automatisch mit 0 initialisiert sind. Diese Annahme ist implementierungsabhängig und kann ggf. durch eine explizite Initialisierung mit *DataWrite* ersetzt werden, die analog zu *Service* 3.11 definiert sind.

Der Anfangszustand s wird über folgenden *DataWrite* gesetzt:

$$DataWrite(\text{„SchreibeZustand\_}s\text{“}) := \begin{cases} driver & = \text{„Memory“}, \\ context & = \text{„Zustand“}, \\ modifier & = Set, \\ value & = s \end{cases} \tag{3.12}$$

Ausführung der Turingmaschine

Die einzelnen Definitionen der Überführungsfunktion (siehe *Service* 3.9) werden in einem *Repetitive* zusammengeführt, wobei *Conditional* (δ_1), *Conditional* (δ_2), ... *Conditional* (δ_n) für alle definierten Stellen der Überführungsfunktion stehen (definiert durch *Service* 3.9).

$$Repetitive(\text{„Turingmaschine“}) := \begin{cases} interval & = 10 \\ duration & = 1 \\ execution\ list & = \{ \\ & Conditional(\text{„}\delta_1\text{“}), \\ & < wait\_* >, \\ & Conditional(\text{„}\delta_2\text{“}), \\ & < wait\_* >, \\ & \vdots \\ & Conditional(\text{„}\delta_n\text{“}), \\ & \} \end{cases} \tag{3.13}$$

Nach dem Start der initialisierenden Dienste 3.11 und 3.12 wird Dienst 3.13 gestartet.

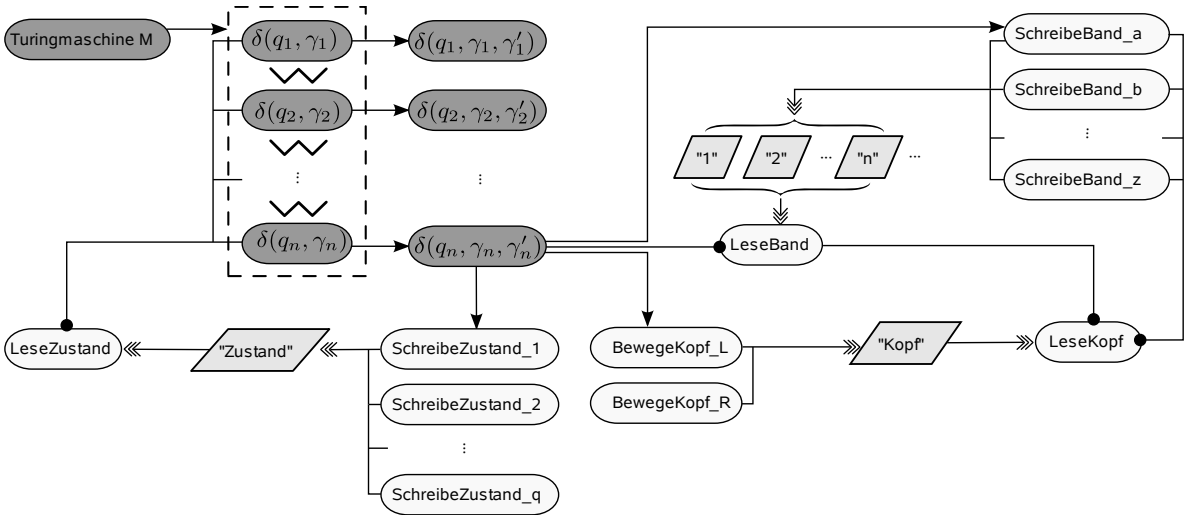

Abbildung 3.9: Gesamtschau der aus TALASSA-Diensten modellierten Turingmaschine M

Dieser läuft gemäß seiner Attribute endlos und führt die für die Überführungsfunktion zuständigen Dienste 3.9 aus. Diese Dienste lesen gemäß ihrer Definition den Zustand sowie die Zeichen des Bandes und schreiben entsprechend Folgezustand und Ausgabezeichen auf das Band. Um Effekte durch parallele Ausführung der *execution list* zu vermeiden, werden die einzelnen Dienste synchronisiert (durch „<wait_*>“) abgearbeitet. Wird schließlich einer der Endzustände $f \in F$ erreicht, bleibt (da kein *Conditional* mehr zutrifft) die Maschine in diesem Zustand stehen. Abb. 3.9 zeigt in einer Gesamtschau die aus Diensten modellierte Turingmaschine M. Die Dienstausführung des Dienstes *Repetitive* („Turingmaschine“) läuft zwar nach obiger Definition unendlich weiter, stellt aber keinen Widerspruch zur Terminierung dar, da ein Verbleiben im Endzustand als Terminierung gilt. Die Terminierung der eigentlichen Dienstausführung könnte selbstverständlich durch zusätzliche *Conditional*-Dienste erreicht werden, die im Falle eines Zustandes $f \in F$ den *Repetitive* („Turingmaschine“) durch einen *stop*-Befehl anhält.

Durch den Nachweis der Turing-Mächtigkeit und unter Annahme der Gültigkeit der Church'schen These [Sch97] sind somit alle „intuitiv berechenbaren Funktionen“ durch TALASSA-Dienste modellierbar.

3.3 Datenorientierte Sensornetze mit Pan

Im bisher beschriebenen Talassa kann der Kontrollfluss einer verteilten Sensornetzanwendung modelliert werden. Durch die Definition verschiedener Dienste und deren Komposition zu komplexen Einheiten werden verteilte Anwendungen in beliebiger Komplexität modellierbar. Dienste auf verschiedenen Sensorknoten, die somit eine verteilte Anwendung darstellen, werden durch Kontrolldaten gesteuert (z. B. starten eines Dienstes, stoppen, etc.). Während der Kontrollfluss in beliebiger Komplexität modellierbar ist, gilt dies im bisher beschriebenen Talassa nicht in gleichem Maße für Nutzdaten.

Durch den primitiven Diensttyp *DataRead* können mithilfe eines Treibers Nutzdaten in beliebiger Form generiert werden, die durch die anderen Diensttypen konsumiert werden. Allerdings ist es nicht möglich, einmal generierte Daten auf ihrem Weg zum Konsumenten zu verändern, sie mit anderen Daten zu aggregieren, beziehungsweise sie *im Netz zu verarbeiten.* Zwar ist es möglich, dass Treiber für primitive Diensttypen jeweils für spezielle Aufgaben/Anwendungen geschrieben werden. Allerdings sind diese Treiber außerhalb von Talassa implementiert und damit nur auf bestimmten Knoten vorhanden. Sie sind in diesem Sinne „spezielle“ physikalische Sensoren und Aktoren, die Nutzdaten aufnehmen (über *DataRead* angesprochen) oder Daten generieren können (über *DataWrite* angesprochen). Diese „speziellen“ physikalischen Sensoren und Aktoren beziehen ihre Funktionalität aus den Treibern, die in einer frei zu wählenden Programmiersprache implementiert sind. Sie sind aus diesem Grund aber Sensorknotenspezifisch, d. h. eine spezielle Eigenschaft derjenigen Sensorknoten, die diese Treiber-Implementierung besitzen. Bezogen auf ein Aggregationsszenario, wo aus Roh-Temperaturdaten der Mittelwert bestimmt werden soll, wären die Aggregationspunkte auf Knoten beschränkt, die diesen speziellen Aggregationstreiber zur Verfügung stellen. Insbesondere müssen die *Daten-verändernden* Treiber für jede Anwendung, beziehungsweise für jede gewünschte Datenverarbeitung, gesondert implementiert werden.

Im Folgenden wird die Datenverarbeitung in Sensornetzen allgemein analysiert und ein Konzept zur flexiblen Datenmodellierung vorgestellt (Pan für engl. ***P****rocess and* ***A****ggregate* ***N****amed data*). Im Anschluss wird dieses Konzept mit dem dienstorientierten Konzept kombiniert und in Talassa integriert.

3.3.1 Datenverarbeitung in Sensornetzen

Viele Anwendungen in Sensornetzen basieren auf der Tatsache, dass Nutzdaten *innerhalb* der Anwendung und *innerhalb* des Netzes verändert werden können. Dies dient insbesondere dazu, Sensorrohdaten zu *aggregieren* oder zu *reduzieren*, um damit die Energiekosten zum Verschicken von Daten zu minimieren. Aggregation wird benutzt, um aus mehreren Eingangsdaten durch eine mathematische Funktion ein einzelnes Datum zu bilden (z. B. Aggregation mehrerer Daten zu deren Maximum, Mittelwert, etc.), das Aggregat genannt wird. Reduktion wird benutzt, um aus einem Rohdatum die gewünschte Information zu extrahieren (z. B. Reduktion aus einem kontinuierlichen Temperaturwert zu einer Ein-Bit-Information *heiß* beziehungsweise *kalt*). Beide Verfahren ermöglichen es, in Sensornetzen Kosten für Kommunikation zu senken, indem die Anzahl (Aggregation) oder die Größe (Reduktion) der verschickten Daten minimiert werden. Beide Verfahren können auch kombiniert werden, um aus beliebigen Sensorrohdaten anwendungsspezifische Nutzdaten abzuleiten.

Aggregation

Reduktion

Beispiel: *Beispielsweise könnte ein Sensor die Anwesenheit von Personen in einem Raum feststellen, ein zweiter die Temperatur messen: in einer Kombination von Aggregation und Reduktion könnte das Datum „Personen anwesend: ja“ und „Temperatur: größer 50 °C“ zum anwendungsspezifischen Datum „Gefahr: ja“ abgeleitet werden.*

Anwendungsspezifische Daten sind in diesem Sinne eine Neudefinition („Kreation“) einer *Größe*. Dies ist nicht notwendigerweise eine physikalische Größe, wie das Beispiel der Neudefinition der Größe „Gefahr“ darlegt.

Abstrakt betrachtet muss ein System für *In-Network-Processing* eine Transformation T ausführen, die aus definierten Eingangsdaten E über eine Überführungsfunktion f ein Ausgangsdatum erzeugt:

$$T = \begin{cases} f : E \times E \times \ldots \times E \rightarrow A \\ (e_1, e_2, \ldots, e_n) \rightarrow f(e_1, e_2, \ldots, e_n) \end{cases}$$

Die Eingangsdaten E sind Nutzdaten, die im Sensornetz selbst generiert werden. Diese können durch Messung physikalischer Sensoren gewonnen werden, anwendungsbezogene Parameter sein, oder Ausgangsdaten anderer Transformationen sein. Die Transformation T führt diese Eingangsdaten in ein Ausgangsdatum über. Die Transformation T ist dabei nicht auf einen einzigen Operator beschränkt (z. B. arithmetische Standardoperatoren wie Addition, Multiplikation, etc. oder Aggregierungsfunktionen wie Maximum, Minimum, Mittelwert, etc.), sondern kann beispielsweise auch mehrere Operatoren kombinieren (z. B. boole'sche Operatoren). Neben der Überführungsfunktion f

ist die Auswahl der Eingangsdaten ein entscheidendes Kriterium für die Mächtigkeit der Datenmodellierung.

Im Folgenden wird eine Modellierung für Nutzdaten vorgestellt, die eine Beschreibung für Transformation ermöglicht. Wie bei der Modellierung des Kontrollflusses bei Talassa soll eine allgemeine Verwendbarkeit der Nutzdaten-Definition für beliebige Transformationen erreicht werden, ohne dass deren Komplexität die Fähigkeiten eines Sensorknoten übersteigt. Die limitierenden Fähigkeiten sind hier wieder die Speicherkapazität (zur Speicherung der Nutzdatendefinitionen) und die Rechenkapazität (Interpretation und Ausführung der Definition). Eine spezielle Betrachtung der Kommunikation zwischen Sensorknoten oder der hierfür benötigten Energie spielt bei der Modellierung der Datenverarbeitung keine Rolle, da die Transformation eines Datums oder mehrerer Daten stets auf einem Knoten stattfindet.

3.3.2 Aufbau der Nutzdaten

Um Nutzdaten sinnvoll weiterverarbeiten zu können, müssen Daten neben dem eigentlichen Nutzinhalt (der gemessene oder berechnete Wert), auch Metadaten enthalten. Diese werden herangezogen, um die Menge der Eingangsdaten eingrenzen zu können und um aus den Eingangsdaten das Ausgangsdatum zu berechnen. Die Metadaten können dabei wie folgt klassifiziert werden.

- Erzeuger
  - Knoten
  - Dienst
- Lokation
  - geographisch
  - logisch
  - absolut
  - relativ
- Zeit
  - Alter
  - Zeitpunkt der Erzeugung
- Qualität
  - Genauigkeit

  - Korrektheit
  - Repräsentativität
- Typ
  - physikalische Größe
  - anwendungsspezifischer Datentyp

Die Auswahl der benötigten Metadaten hängt dabei vom Anwendungsfall und der zur Datentransformation eingesetzten Funktion ab. Die Metadaten können dazu genutzt werden, die Eignung der Eingangdaten für die Transformation T auszuwerten. Beispielsweise kann der Ursprungsort (z. B. geographischer oder logischer Ort einer Temperaturmessung) eines Datums genutzt werden, um nur Daten aus einer bestimmten Region zu einem Mittelwert zu verarbeiten (z. B. die Durchschnittstemperatur eines Raums mit mehreren Temperatursensoren). Die Einschränkung der Eingangsdaten kann über eine beliebige Kombination von Nebenbedingungen über die Metadaten erfolgen.

Erzeuger

Lokation

Zeit

Qualität

Der *Erzeuger* des Datums ist dabei oft von entscheidender Bedeutung. Sowohl dessen Kennung (entweder knotenbezogen über die Knotenadresse oder funktionsbezogen über die Dienstkennung) als auch dessen Lokation sind dabei wichtig. Bei der *Lokation* kann zudem unterschieden werden zwischen einer geographischen Position (beispielsweise GPS-Daten)oder einer logischen Angabe (beispielsweise „Büro 363“), die beide wiederum absolut oder relativ (beispielsweise „im 10m-Umkreis von Knoten X“ oder „alle Nachbarbüros von Büro 363“) erfolgen können. *Zeit*bezogene Metadaten sind von Bedeutung, wenn die Aktualität der erhobenen Daten relevant ist und deren Erhebung mit Kosten (zum Versand oder zur Messung) verbunden sind. Unter *Qualität* werden Metadaten subsumiert, die die Beschaffenheit der Daten bezüglich ihrer Eignung für ein „Ziel“ quantifiziert erfassen. Beispielsweise kann die Genauigkeit die relative Messungenauigkeit eines (kontinuierlichen oder diskret geordneten) Messwertes angeben, die Korrektheit die Abschätzung der Richtigkeit einer Angabe, die Repräsentativität die Anzahl der verwendeten Quellen (und damit die Möglichkeit zu weiterer Verarbeitung). Letztgenanntes Metadatum kann beispielsweise für eine schrittweise Ermittlung eines Mittelwerts genutzt werden, indem jeder Mittelwert gleichzeitig die Information mitführt, aus wie vielen Ursprungswerten dieser berechnet worden ist. Nachfolgende Aggregationen können somit vorberechnete Mittelwerte zu einem Gesamt-Mittelwert zusammenfassen. Der Typ eines Sensordatums stellt die physikalische Größe oder das anwendungsspezifische Pendant dar (siehe obiges Beispiel zum anwendungsspezifischen Datentyp „Gefahr“).

Nutzdatum	
source service	Sendender Dienst/Knoten
value	Wert (durch Sensor gemessen oder aus anderen Daten berechnet)
age	Alter des (gemessenen/berechneten) Wertes
accuracy	Genauigkeit des (gemessenen/berechneten) Wertes
num_sources	Anzahl der Quellen, die zur Ermittlung des Wertes herangezogen wurden
dataID	Kennung der Datendefinition

Tabelle 3.12: Beispielhafte Auswahl von (Meta-)Attributen eines Nutzdatums

Die Festlegung der Attribute des Nutzdatums hat große Auswirkungen auf die spätere Funktionalität des Sensornetzes. Anhand dieser Attribute kann die Menge der Eingangsdaten wie dargelegt mit Nebenbedingungen eingeschränkt werden. Gleichzeitig werden diese Attribute zur weiteren Verarbeitung der Daten genutzt (z. B. zur schrittweisen Berechnung des Mittelwerts).

Eine flexible, im Nachhinein erweiter- und änderbare Beschreibung der Nutzdaten ist wünschenswert, stellt die teilnehmenden Knoten jedoch vor das Problem, dass sämtliche Implementierungskomponenten, die die Nutzdaten betreffen, variabel gestaltet werden müssen. Dies betrifft insbesondere das Senden und Empfangen von Nutzdatenpaketen und die Weiterverarbeitung von Nutzdaten. Sind die Attribute des Nutzdatums dagegen *a priori* festgelegt, können sowohl das Senden wie auch das Empfangen der Nutzdatenpakete durch optimierte, fest implementierte Programmroutinen ausgeführt werden. Genauso profitiert eine Implementierung zur Datenverarbeitung von einer festgelegten Attributmenge, da keine dynamisch definierten Attribute interpretiert werden müssen. Es wird deswegen davon ausgegangen, dass der Aufbau der Nutzdaten zwar *vor* der Ausbringung der Sensorknoten frei festgelegt (d. h. frei implementiert) werden kann, dieser aber sich für die Lebenszeit eines Sensorknoten beziehungsweise eines Sensornetzes nicht mehr ändert. Es gilt aber weiterhin, dass Nutzdaten und ihre Verarbeitung auch während der Lebenszeit des Sensornetzes neu definiert werden können. Der Inhalt und die Verarbeitung der Nutzdaten sind frei definierbar, lediglich die Meta-Attribute sind festgelegt.

Definition eines Datums	
dataID	Kennung der Datendefinition
<Produktionsvorschrift>	Festlegung der Produktion des Datums

Tabelle 3.13: Allgemeiner Aufbau einer Datendefinition für ein Nutzdatum

Eine beispielhafte, für ein reales Sensornetz sinnvolle Auswahl von Metadaten (Attributen) für ein Nutzdatum zeigt Tab. 3.12.[4] Das Attribut *source service* gibt hier den Erzeuger an und *value* den eigentlichen Wert des Sensordatums. Im Falle der Nutzung eines dienstorientierten Systems beinhaltet *source service* die Dienstkennung des sendenden Dienstes. Die Metadaten *age* und *accuracy* geben das Alter beziehungsweise die relative Genauigkeit des Wertes an. Mit *num_sources* wird angegeben, wie viele Ausgangsdaten zur Produktion des Wertes benutzt wurden. Genau eine Quelle wurde für dieses Metadatum gesetzt, wenn der Wert in einer Messung durch einen physikalischen Sensor gewonnen wurde; mehrere Quellen werden dann gesetzt, wenn es sich um ein aggregiertes Datum handelt (die Anzahl der verwendeten Quellen, beziehungsweise die Summe der *num_sources* aller verwendeten Quellen wird dann als Wert gesetzt). Die Kennung der Datendefinition (*dataID*) bezeichnet den Typ des Datums (beispielsweise die physikalische Größe des Wertes).

Zur Weiterverarbeitung dieses Datums muss eine Transformation definiert werden, welche die Eingangsdaten festlegt und die Funktion, die das Ausgabedatum erzeugt. Der folgende Abschnitt führt dazu die Datendefinition ein, die aus dem oben eingeführten *abstrakten* Bauplan des Nutzdatums *konkrete* Daten generiert.

3.3.3 Datendefinition

Datentypen: primitiv und abgeleitet

Analog zu *primitiven* und *komponierenden* Diensten bei TALASSA existieren bei PAN *primitive* und *abgeleitete* Daten. Primitive Daten werden von geeigneten Knoten, die über bestimmte physikalische beziehungsweise virtuelle Sensoren verfügen, generiert. Diese Daten werden von Treiberimplementierungen gemessen oder gelesen. Sie sind damit die zur Verfügung stehenden Urdaten in einem Sensornetz. Aus diesen primitiven Daten können durch Transformation abgeleitete Daten entstehen, welche wiederum

[4] Anm.: Das dargestellte Datum mit seinen Attributen stellt zwar nur ein Beispiel dar, welches bei Bedarf geändert oder ergänzt werden kann, wird aber hier als für die meisten Sensornetze ausreichend angesehen.

bei Bedarf weiter transformiert werden können. Jedes Datum, das in einem datenorientierten Sensornetz als Nutzdatum verwendet wird, muss (wiederum analog zur Dienstdefinition) über eine Datendefinition beschrieben werden.

In Tab. 3.13 ist der allgemeine Aufbau einer Datendefinition für Nutzdaten angegeben. Die Datendefinition enthält zwei Teile: eine eindeutige Kennung (*dataID*) und die Festlegung auf die Produktion (<Produktionsvorschrift>) des Datums. Die eindeutige Kennung der Datendefinition kann im einfachsten Fall mit einer physikalischen Größe gleichgesetzt werden. Die Produktion des Datums bezieht sich auf die Art und Weise, wie ein Sensordatum gemessen oder berechnet wird. In diesem Teil der Datendefinition wird festgelegt, welcher Treiber genutzt wird oder wie aus Eingangsdaten das Ausgangsdatum abgeleitet wird.

Beispiel: *Werden in einem Sensornetz beispielsweise Temperatur, Helligkeit und Feuchtigkeit gemessen, gibt es drei Datendefinitionen mit jeweils einer Kennung für jede der drei physikalischen Größen. Die Produktionsvorschrift ist jeweils ein Treiber.*

1. • dataID = *„Temperatur“*
   • driver = *<Treiber für physikalischen Temperatursensor>*
2. • dataID = *„Helligkeit“*
   • driver = *<Treiber für physikalischen Helligkeitssensor>*
3. • dataID = *„Feuchtigkeit“*
   • driver = *<Treiber für physikalischen Feuchtigkeitssensor>*

Wird von einem DataRead-*Dienst „MesseTemperatur“ ein Temperaturwert von beispielsweise 20 °C gelesen, generiert dieser folgendes Nutzdatum:*

- source service = *„MesseTemperatur“*
- value = *„20“*
- ...
- dataID = *„Temperatur“*

Dieses Nutzdatum wird dann an den im DataRead-*Dienst angegebenen Empfängerdienst gesendet. Dort kann es gemäß der Attribute als Temperaturwert erkannt und ausgewertet werden.*

Primitive Daten

Die Datendefinitionen für *primitive Daten* ist die Entsprechung für die in Abschnitt 3.2 beschriebene Definition des Dienstes *DataRead*. Während dort der Dienst *direkt* ein Nutzdatum mit der Referenz auf einen Treiber produziert, wird hier die Produktion

Primitives Datum	
dataID	Kennung dieser Datendefinition
driver	Treiberkennung zum Auslesen des (physikalischen/virtuellen) Sensors

Tabelle 3.14: Attribute der Definition eines primitiven Datums

des Nutzdatums über eine Datendefinition *abstrahiert*. Trotzdem bleibt die prinzipielle Vorgehensweise gleich, indem für jeden Sensortyp eine Treiberimplementierung zur Verfügung stehen muss, welche über die Treiberkennung innerhalb der Datendefinition referenziert wird. In Tab. 3.14 sind die dafür notwendigen Attribute dargestellt: die für alle Datendefinitionen obligatorische eindeutige Datenkennung (*dataID*) und die als Produktionsvorschrift geltende Treiberkennung (*driver*). Auch hier beschränkt sich der „Sensor“ und dessen Zugriff über Treiber nicht ausschließlich auf „reale physikalische Sensoren“, sondern kann auf beliebige Datenproduzenten, sogenannte „virtuelle Sensoren“, verallgemeinert werden. Mit virtuellen Sensoren können beliebige Nutzdaten im Sensornetz produziert und genutzt werden, insbesondere können mit diesen auch Variableninhalte ausgelesen und somit dem Sensornetz zur Verfügung gestellt werden. Datendefinitionen erlauben eine Abstraktion von der Daten*produktion* in einer konkreten Anwendung. Insbesondere die Typisierung (wie z. B. oben mit *dataID*) unterstützt eine Kapselung der (Nutz-)Datendefinition von der Anwendungsdefinition (Kontrollfluss) und erlaubt eine Weiterverarbeitung dieser Daten. Ohne eine Abstraktion der Datenproduktion (und ohne Typisierung) können Daten lediglich aufgrund ihrer Herkunft unterschieden werden (z. B. Temperaturdienst A) ohne die Möglichkeit „gleiche“ Daten herkunftsunabhängig weiterzuverarbeiten (Temperaturdaten, unabhängig ihrer Herkunft von Temperaturdienst A, B oder C).

Abgeleitete Daten

Die flexible Weiterverarbeitung von Daten ist eine entscheidende Fähigkeit eines Sensornetzes, in dem Anwendungen berechnete Daten (z. B. durch Aggregation) nutzen. Wenn zusätzlich Änderungen in der Anwendung erwartet werden oder weitere Anwendungen nachträglich ins Sensornetz integriert werden sollen, muss die Weiterverarbeitung auch änderbar und erweiterbar sein.

Die Weiterverarbeitung in dem hier vorgestellten Konzept zur Datendefinition geschieht durch *abgeleitete Daten*. Entsprechend den obigen Überlegungen müssen abgeleitete Daten eine Menge von Eingangsdaten durch eine Überführungsfunktion in ein Ausgangsdatum transformieren. Die grundlegende Definition besteht deswegen aus drei Teilen: die obligatorische Datenkennung (*dataID*), den Eingangsdaten (<Eingangs-

Abgeleitetes Datum	
dataID	Kennung dieser Datendefinition
<Eingangsdaten>	Beliebig viele Eingangsdaten (siehe Tab. 3.16)
<Überführungsfunktionen>	Überführungsfunktion (siehe Tab. 3.17) für jedes Attribut des Nutzdatums

Tabelle 3.15: Attribute der Definition eines abgeleiteten Datums

Eingangsdaten	
dataID	Kennung der Datendefinition der Eingangsdaten
min	Minimal notwendige Anzahl an Eingangsdaten
max	Maximal zulässige Anzahl an Eingangsdaten
sourceID	Kennung des Eingangsdatums zur Weiterverwendung in der Überführungsfunktion (siehe Tab. 3.17)

Tabelle 3.16: Definition der Eingangsdaten für ein abgeleitetes Datum

daten>) und der Überführungsfunktion (<Überführungsfunktionen>). Tab. 3.15 gibt diesen grundlegenden Aufbau der Definition eines abgeleiteten Datums wieder. Die Eingangsdaten werden mithilfe der Datenkennung referenziert, wobei gleichzeitig die Anzahl der Daten der entsprechenden Kennung angegeben wird (siehe Tab. 3.16). Jede Menge von Eingangsdaten erhält in der Definition der Eingangsdaten wiederum eine spezielle Kennung, welche in der Überführungsfunktion benutzt wird, um diese als Parameter referenzieren zu können. Im Unterschied zur Datenkennung, welche eindeutig ist und global für das gesamte Sensornetz gilt, hat die Kennung der Eingangsdaten nur lokale Bedeutung innerhalb einer Datendefinition.

Beispiel: *Greift man das Beispiel von Abschnitt 3.3.1 auf mit den primitiven Daten „Personen anwesend?“ und „Temperatur“, kann das Ausgangsdatum „Gefahr“ abgeleitet werden. Die Datendefinition wäre dann wie folgt:*

- dataID = *„Gefahr“*
- *<Eingangsdaten>*
  1. – dataID = *„Personen anwesend?“*
     – min = *1*
     – max = *1*
     – sourceID = *„Personen anwesend? #1“*

Überführungsfunktion	
Attribut 1	Operator Parameter 1 (referenziert über *sourceID* der Eingangsdaten) ... Parameter m (referenziert über *sourceID* der Eingangsdaten)
...	...
Attribut n	Operator Parameter 1 (referenziert über *sourceID* der Eingangsdaten) ... Parameter m (referenziert über *sourceID* der Eingangsdaten)

Bem.: Parameter können selbst wiederum aus Operator und Parametern aufgebaut sein.

Tabelle 3.17: Definition der Überführungsfunktion für ein abgeleitetes Datum

2. – dataID = *„Temperatur“*
– min = *1*
– max = *1*
– sourceID = *„Temperatur #1“*

- *<Überführungsfunktionen> = siehe nächstes Beispiel*

Um ein abgeleitetes Datum „Gefahr“ zu bilden, müssen demnach genau ein Eingangsdatum „Personen anwesend?“ und genau ein Eingangsdatum „Temperatur“ vorhanden sein. Diese beiden Eingangsdaten werden dann in einer Überführungsfunktion zum Datum „Gefahr“ abgeleitet. In dieser Überführungsfunktion wird auf diese beiden Eingangsdaten mit den willkürlich vergebenen Namen „Personen anwesend? #1“ und „Temperatur #1“ referenziert.

In der Definition der allgemeinen Überführungsfunktion (siehe Tab. 3.17) werden alle Attribute des Nutzdatums durch die Angabe einer separaten Überführungsfunktion definiert. Die Attribute beziehen sich dabei auf die *a priori* getroffene Festlegung des Nutzdatums (siehe z. B. Tab. 3.12). Tab. 3.17 zeigt den Aufbau der vollständigen Überführungsfunktion über Operatoren und deren Parameter. Die *Operatoren* der Übergangsfunktion stellen im Gegensatz zu sämtlichen anderen Definitionsteilen eines abgeleiteten Datums (wie Eingangsdaten und Parameter der Funktion) eine *statische* Komponente dar. Da es nicht möglich ist, beliebige Berechnungs- oder Ableitungsvorschriften effizient und dynamisch an Sensorknoten zu verteilen, sind die Operatoren,

ähnlich den Treibern, festgelegte Implementierungen. Steht beispielsweise die Fakultätsfunktion nicht als Operator zur Verfügung, besteht keine Möglichkeit, die Berechnungsvorschrift für diesen Operator nachträglich an die Sensorknoten zu senden.

Die Entscheidung für die zur Verfügung stehende Menge von Operatoren hängt von der Leistungsfähigkeit der Sensorknoten und der gewünschten Datenverarbeitung ab. Die folgende Aufzählung zeigt eine beispielhafte, nicht vollständige Liste von möglichen Operatoren:

- mathematische Grund-Operatoren
  - Addition
  - Subtraktion
  - Multiplikation
  - Division
- Aggregat-Operatoren
  - Minimum
  - Maximum
  - Summe
  - Mittelwert
- vergleichende Operatoren
  - größer
  - kleiner
  - gleich
  - ungleich
- aussagenlogische Grund-Operatoren
  - und
  - oder
  - nicht
- erweiterte aussagenlogische Operatoren
  - Implikation

Die obige Auswahl an Operatoren ermöglicht durch Kombination eine Vielzahl an vorstellbaren Transformationen.

Beispiel: *Greift man das Beispiel von Abschnitt 3.3.1 auf mit den primitiven Daten „Personen anwesend“ (als boole'sche Angabe interpretierte Ganzzahl) und „Temperatur“*

(als Grad Celsius interpretierte Ganzzahl), kann das Ausgangsdatum „Gefahr“ (als boole'sche Angabe interpretierte Ganzzahl) wie folgt abgeleitet werden:

Operator „Implikation“		
Parameter	*Operator „Und“*	
	Parameter	*Operator „Größer“*
		Parameter „Temperatur #1“
		Parameter 50
	Parameter	*Operator „Gleich“*
		Parameter „Personen anwesend? #1“
		Parameter 1 (für boole'sch „wahr“*)*
Parameter	*1*	
Parameter	*0*	

Von innen nach außen gelesen wird überprüft, ob die Temperatur größer als 50° C ist (Operator „größer“), Personen im Raum anwesend sind (Operator „gleich“); ob beide vorige Bedingungen zutreffen (Operator „und“); und schließlich in Abhängigkeit der vorangegangenen Evaluation die Zuweisung 1 bei Gefahr oder 0 bei keiner Gefahr (Operator „Implikation“). Die Eingangsdaten „Personen anwesend“ und „Temperatur“ werden auf diese Weise zu einer Ein-Bit-Information aggregiert und reduziert.

Bei der Implementierung eines Sensornetzes ist es für die Flexibilität von entscheidender Bedeutung, welches Operatorensystem ausgewählt wird, d. h. welche der obigen Operatoren von den Sensorknoten „angeboten“ beziehungsweise implementiert werden. Werden zu wenige beziehungsweise ungeeignete (Grund-)Operatoren angeboten, kann die Konstruktion „anwendungsspezifischer“ Operatoren sehr komplex werden. Ein fehlender Multiplikations-Operator könnte beispielsweise durch verkettete Addition gebildet werden, würde aber zu langen, und damit ineffizienten Operatorkonstruktionen führen. Im Allgemeinen existiert *kein* Operatorsystem, das die Konstruktion sämtlicher vorstellbarer Operatoren erlaubt. So bietet das in der vorangegangenen Aufzählung eine große Flexibilität, kann jedoch nicht beliebige Operatoren (wie z. B. Sinusfunktion, Logarithmusfunktion, etc.) nachbilden. Durch die nachgewiesene Turing-Mächtigkeit von TALASSA kann zwar allein durch Dienste prinzipiell jede „intuitiv berechenbare Funktion“ modelliert werden. Allerdings sagt die Turing-Mächtigkeit nichts über die Komplexität der notwendigen Modellierung aus. Zentrale (oft gebrauchte) Operatoren sollten deswegen aus Effizienz- und Flexibilitätsgründen als Implementierungen auf den Sensorknoten zur Verfügung stehen.

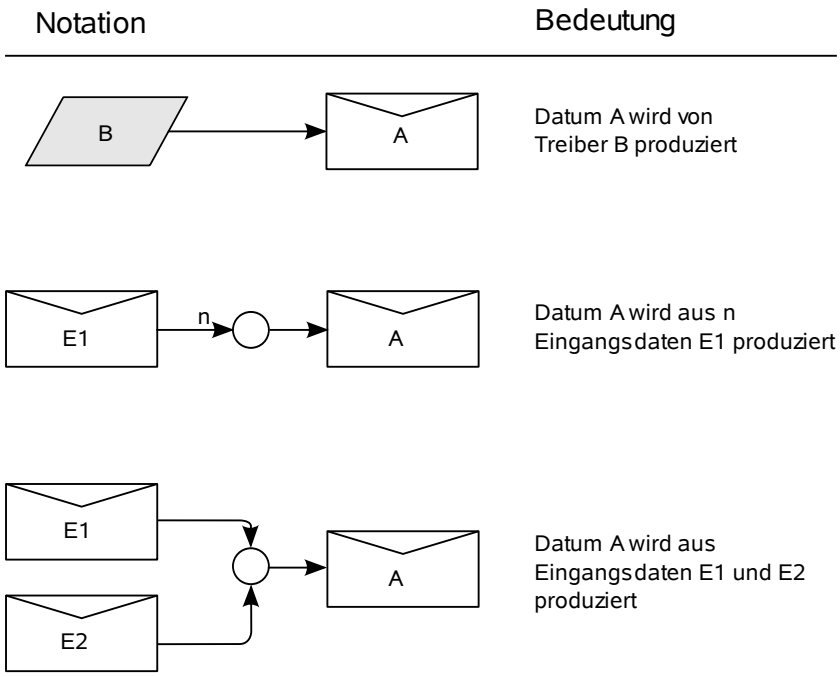

Abbildung 3.10: Graphische Repräsentation primitiver (links) und abgeleiteter Daten (rechts)

3.3.4 Notation

In Analogie zur graphischen Notation von Diensten wird hier eine Notation für Daten eingeführt. Die Nutzdaten von Pan werden, unabhängig von ihrer Produktion, als symbolisierter Briefumschlag mit der entsprechenden Kennung dargestellt.

Der Treiber beziehungsweise die zur Produktion verwendeten Eingangsdaten werden wie in Abb. 3.10 dargestellt. Die Darstellung der Regeln, als Kombination von Überführungsfunktion, Eingangsparameter und resultierendem Attribut des Ausgangsdatums, findet sich in Abb. 3.11.

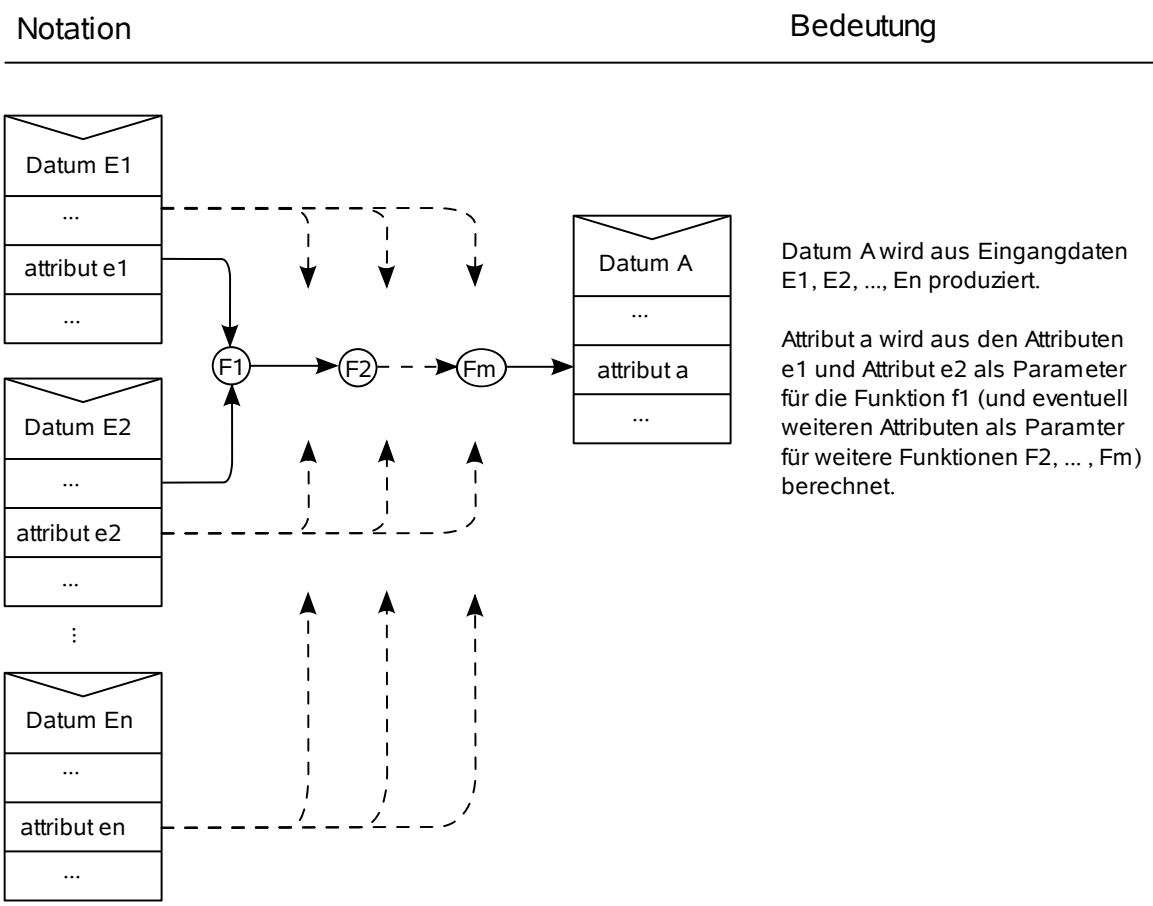

Abbildung 3.11: Graphische Repräsentation der Überführungsfunktion

3.4 Kombination von Talassa-Diensten und Pan-Daten

In diesem Abschnitt wird erläutert, wie die Dienste von TALASSA und die Daten von PAN kombiniert werden. Das resultierende PANTALASSA vereint damit den dienstorientierten Ansatz von TALASSA mit dem datenorientierten Ansatz von PAN.

Nutzdaten werden im bisher beschriebenen TALASSA durch die Treiber der entsprechenden *DataRead*-Dienste generiert. Diese sind damit auch die einzigen *Datenquellen*, sämtliche andere Diensttypen sind *Datensenken*. In Abb. 3.12 ist die Verknüpfung zwischen der produzierenden Einheit Treiber und Datenquelle (*DataRead*), die das Nutzdatum generiert, und den konsumierenden Diensttypen (*Repetitive*, *Conditional*, *Event* und *DataWrite*) dargestellt. Insbesondere fungiert der *DataRead*-Dienst nicht nur

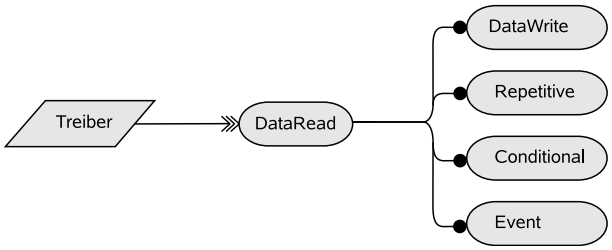

Abbildung 3.12: Produzenten und Konsumenten von Nutzdaten

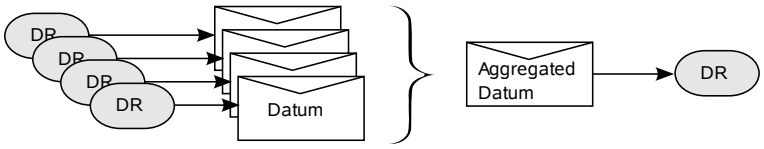

Abbildung 3.13: Kombination von *DataRead* und Datendefinition

als alleinige Datenquelle, er ist auch der einzige Diensttyp, der für andere Dienste als Datenquelle vorgesehen ist. Abb. 3.12 folgend ist die einzige Möglichkeit, Nutzdaten zu verändern, ein spezieller Treiber für den Diensttyp *DataWrite*. Von dieser Möglichkeit wurde beispielsweise in Abschnitt 3.2.5 Gebrauch gemacht, um bei virtuellen Aktoren Rechenoperationen zu erlauben. Allerdings sind allgemeine Übergangsfunktionen mit mehreren Eingangsdaten, insbesondere bei Berücksichtigung spezieller Nebenbedingungen, nicht modellierbar. Die Lösung dieses Problems ist die Kombination des dienstorientierten mit dem datenorientierten Konzept.

Im dienstorientierten Konzept werden Nutzdaten von dem Diensttyp *DataRead* produziert, und im datenorientierten Konzept können Eingangs-Nutzdaten zu beliebigen Ausgangs-Nutzdaten weiterverarbeitet werden. Die Kombination beider Konzepte eröffnet somit die Möglichkeit, dass die Nutzdaten eines bzw. beliebig vieler *DataRead*-Dienste als Eingangsdaten für ein abgeleitetes Datum dienen, welches wiederum als Nutzdatum eines *DataRead* gilt. Abb. 3.13 zeigt diesen Zusammenhang. Somit dient der Diensttyp *DataRead* als Quelle für Eingangsdaten, die mit den Regeln der Datendefinition weiterverarbeitet werden, und das resultierende Ausgangsdatum wird wiederum vom Diensttyp *DataRead* „produziert“. Die Definition des *DataRead*-Dienstes bei der kombinierten Anwendung beider Konzepte ändert sich dabei wie folgt: statt der Kennung des verantwortlichen Treibers wird die Kennung der verantwortlichen Datendefinition angegeben (z. B. statt *driver* =„Helligkeitssensor“ steht *dataID* =„Helligkeit“).

Die oben beschriebenen Methode erlaubt die Verbindung zwischen Dienstbeschreibung in Talassa und Datenbeschreibung in Pan. Während in Talassa der Kon-

trollfluss durch Dienste beschrieben wird, ist die Generierung und sensornetzinterne Verarbeitung der Nutzdaten durch PAN beschreibbar. Durch die Verwendung der durch PAN beschriebenen Daten in *DataRead*-Diensten gelingt die Kombination der Vorteile von dienst- und datenorientierten Sensornetzen auf geradlinige Weise. Die durch den Zusammenschluss der beiden Sprachen entstandene vereinigende Beschreibungssprache wird PANTALASSA genannt.

3.5 Umsetzung

Im folgenden Kapitel wird die konkrete softwaretechnische Umsetzung der abstrakten PanTalassa-Architektur beschrieben. Zuerst wird der Aufbau einer virtuellen Maschine TalassaVM ohne Bezug auf Pan-Beschreibungen vorgestellt. Darauf folgend wird gezeigt, wie sich der Pan in die bestehende TalassaVM integrieren lässt und damit eine vollständige virtuelle Maschine PanTalassaVM realisieren lässt. Zum Abschluss wird die Realisierung der Knoten-Dienst-Zuordnung erläutert.

3.5.1 TalassaVM

In diesem Abschnitt wird zunächst die Architektur der virtuellen Maschine TalassaVM dargestellt. Im Anschluss folgen Erläuterungen zum Ablauf der verschiedenen Diensttypen in der virtuellen Maschine.

Architektur und Schnittstellen der TalassaVM

Die zentralen Komponenten der Softwarearchitektur der virtuellen Maschine von Talassa sind die vier Komponenten *Message*, *Timer*, *Service executor* und *Scheduler*. In Abb. 3.14 ist ein Überblick über alle Komponenten und ihre Beziehen dargestellt.

Message

Die Komponente *Message* stellt die Anbindung der TalassaVM an die Netzwerkschicht dar. Diese Komponente sendet und empfängt Sensor- und Steuerdaten. Sensordaten werden über die Komponentenschnittstelle *datum* von einem knoteneigenen *DataRead* ausgelesen, beziehungsweise von einem knotenfremden *DataRead* empfangen. Steuerdaten dienen dem Starten und Stoppen von Diensten und werden über die Schnittstellen *start/stop service* und *fire event* gesendet, und über die Schnittstellen *start/stop* und *event* empfangen. Die Start- und Stop-Pakete beinhalten dabei die Kennung *serviceID* des entsprechenden Dienstes, die Event-Pakete beinhalten die Kennung *eventID* des entsprechenden Events. Die Schnittstelle *send confirm* wird genutzt, um die Ausführung eines Dienstes zu bestätigen. Das Confirm-Paket wird über die Schnittstelle *confirm* empfangen und beinhaltet die Kennung *serviceID* des entsprechenden Dienstes.

Die ankommenden Pakete werden im Falle von Sensordaten über die Schnittstelle *cache message* an die Komponente *Answer cache* geschickt und dort bis zur weiteren

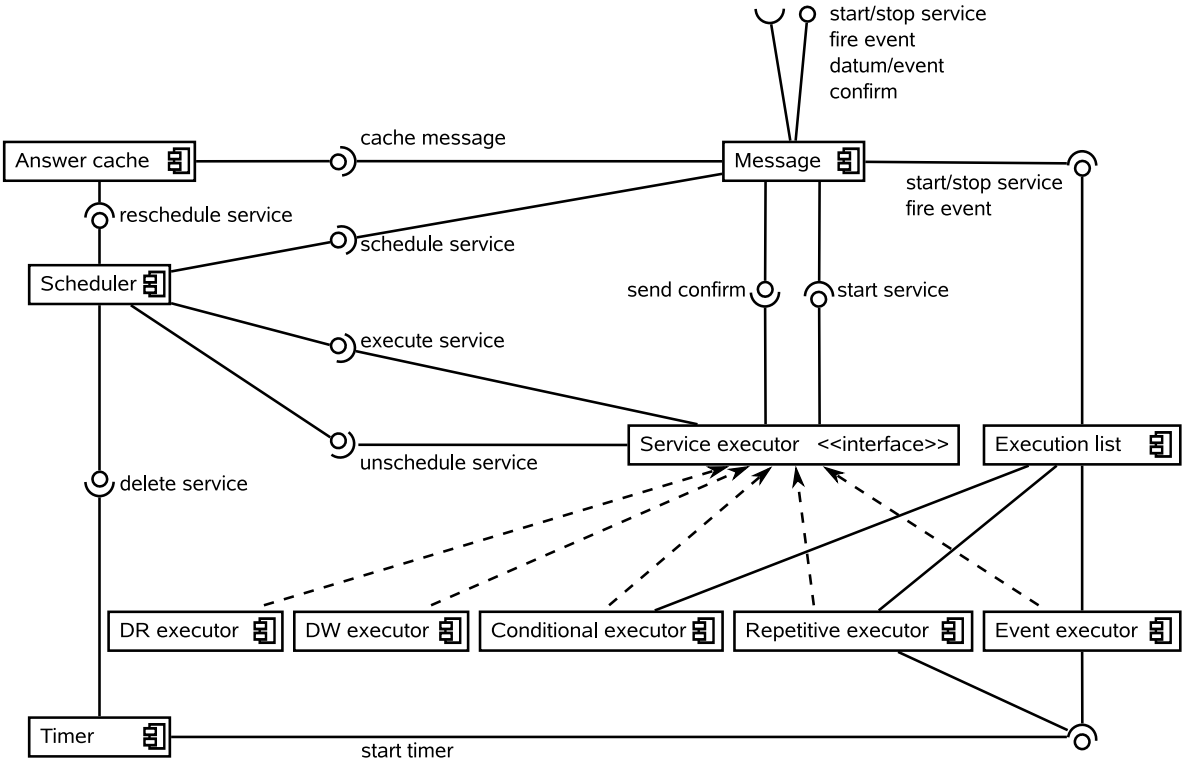

Abbildung 3.14: Komponenten und Softwareschnittstellen der TalassaVM

Verwendung zwischengespeichert. Steuerdaten werden über die Schnittstelle *schedule service* an die Komponente *Scheduler* geschickt.

Scheduler

Die Komponente *Scheduler* übernimmt das Starten und Stoppen der knoteneigenen Dienste. Wird von *Message* oder *Service executor* der Stop eines Dienste verlangt, wird dieser aus der Liste der auszuführenden Dienste gelöscht. Gleichermaßen wird bei einem Start-Wunsch der entsprechende Dienst ans Ende dieser Liste aufgenommen. Die Liste der zu startenden Dienste wird dabei nach der First-in-first-out-Strategie abgearbeitet. Über die Schnittstelle *execute service* werden die Dienste beim passenden *Service executor* gestartet. Eine Änderung der Reihenfolge über Priorisierung ist nicht vorgesehen. Jedoch können einige Dienste nicht sofort gestartet werden, da diese von Parametern abhängig sind. In diesem Fall meldet der *Scheduler* die fehlenden Parameter bei der Komponente *Answer cache* an und startet dann die nächsten Dienste. Sobald die Daten bei der Komponente *Answer cache* eintreffen, wird der entsprechende Dienst über die *reschedule service* erneut dem *Scheduler* übergeben. Wenn ein Dienst von einer synchronisierten Ausführungsliste gestartet wird, bestätigt der *Scheduler* über

send confirm die Ausführung des Dienstes. Dies betrifft den Dienst *Event* und alle Dienste, die einen *DataRead* als Parameter haben.

Service executor

Das Interface *Service executor* wird von den Komponenten *DataRead executor*, *DataWrite executor*, *Event executor*, *Repetitive executor* und *Conditional executor* implementiert. Diese Komponenten sind für die eigentliche Ausführung der entsprechenden Dienste verantwortlich. Die komplexen Dienste *Event*, *Repetitive* und *Conditional* haben ihrer Definition entsprechend eine Ausführungsliste, die über die Komponente *Execution list* abgearbeitet wird. Die *Execution list* übergibt die enthaltenen Dienststeuerungen über die Schnittstellen *start/stop service* und *fire event* an die Komponente *Message*.

Die Komponente *Event executor* setzt bei ihrem Start über die Schnittstelle *start timer* einen Timer, der beim Ablauf der gegebenen Zeit den Dienst aus der Dienstliste der Komponente *Scheduler* löscht.

Allgemeiner Ablauf der Dienstausführung

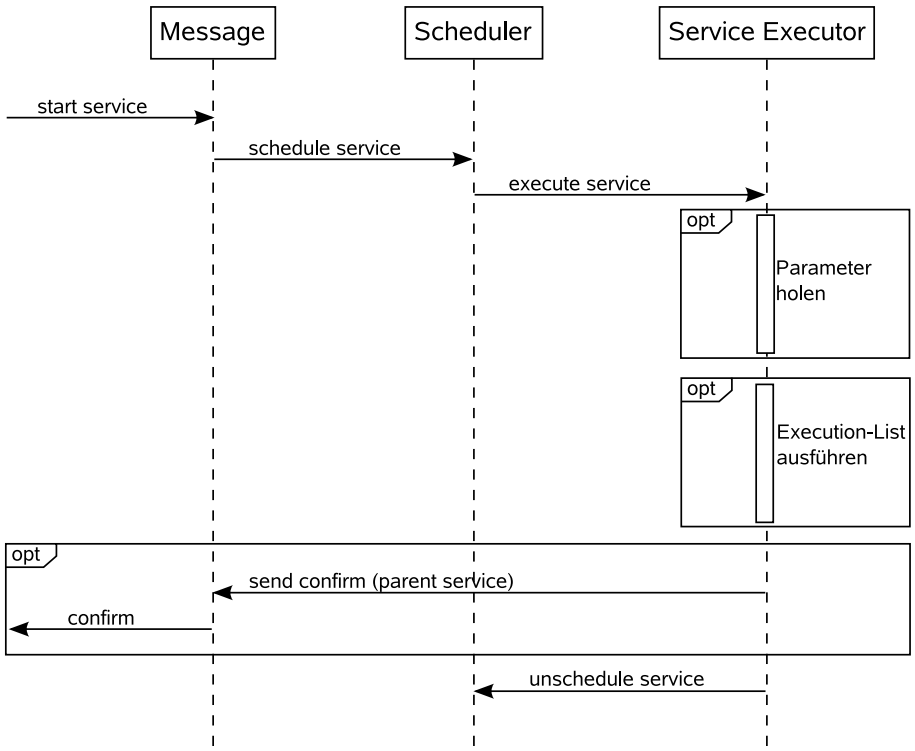

Abbildung 3.15: Allgemeiner Ablauf einer Dienstausführung in der TalassaVM

Der grundlegende Ablauf einer Dienstausführung ist in Abb. 3.15 dargestellt. Über die Komponente *Message* wird eine Start-Aufforderung des Dienstes empfangen. Über *schedule service* wird der *Scheduler* aufgerufen, der wiederum den Dienst über *execute service* den passenden *Service executor* startet. Dieser führt den Dienst aus. Je nach Diensttyp kann der *Service executor* in der Ausführungsphase noch fehlende Parameter holen und eine *execution list* ausführen. Wurde ein Dienst von einer synchronisierten Ausführungsliste gestartet, wird ein *confirm* an den Dienst der aufrufenden Ausführungsliste gesendet. Nachdem der Dienst ausgeführt wurde, wird der Dienst aus der Dienstliste der Komponente *Scheduler* entfernt.

Anfordern von Parametern

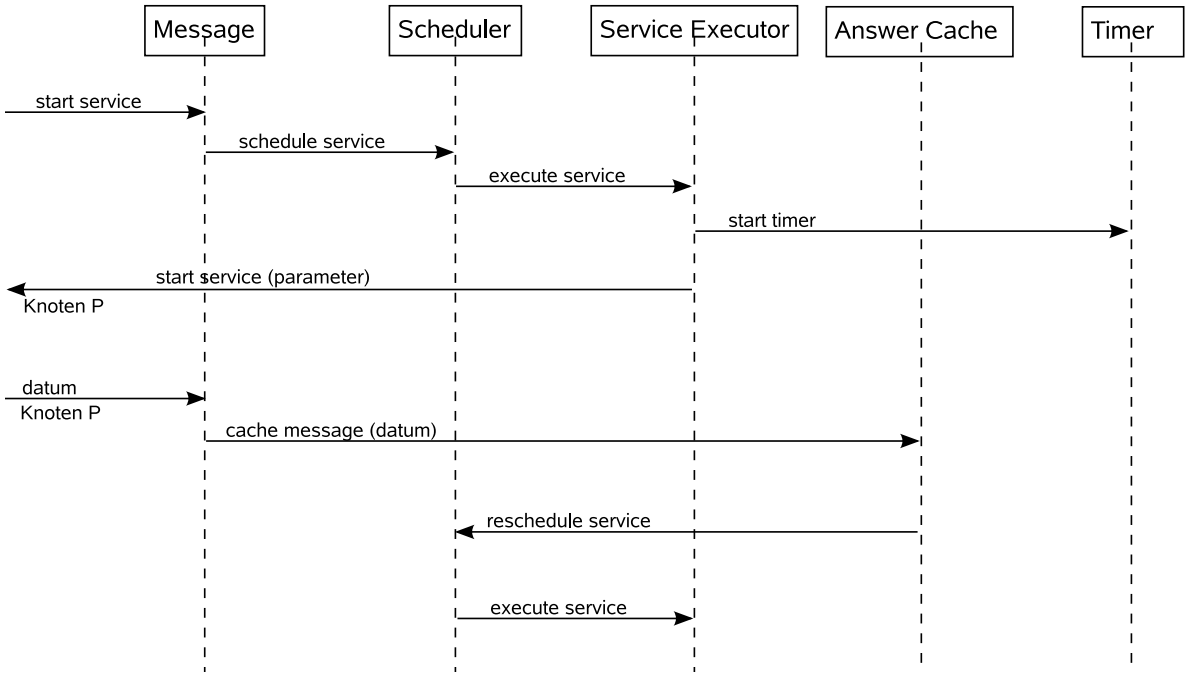

Abbildung 3.16: Allgemeiner Ablauf beim Anfordern von Parametern in der TalasSAVM

Wenn Dienste von Parametern abhängig sind, sind diese über die entsprechenden Parameter-Dienste erreichbar. In Abb. 3.16 ist der allgemeine Ablauf dargestellt, wie Parameter angefordert werden. Nach dem schon zuvor erläuterten Start des Dienstes im *Service executor*, stellt dieser das Fehlen eines oder mehrerer Parameter fest. In der Abb. 3.16 wird der Einfachheit halber von nur einem fehlenden Parameter ausgegangen, der von einem Dienst auf dem Knoten P empfangen werden soll. Nach Definition ist

dieser Parameter ein *DataRead* -Dienst. Dieser wird mit einem *start service* über die Komponente *Message* gestartet.

Sobald das entsprechende Datum von Knoten P in der Komponente *Message* ankommt, wird der *Answer cache* informiert. Dieser wiederum informiert den *Scheduler* über die Ankunft des neuen Datums, der dann den Dienst erneut beim *Service executor* startet. Der Dienst wird dann mit dem nun vorhandenen Parameter ausgeführt.

Ausführung von *DataRead*

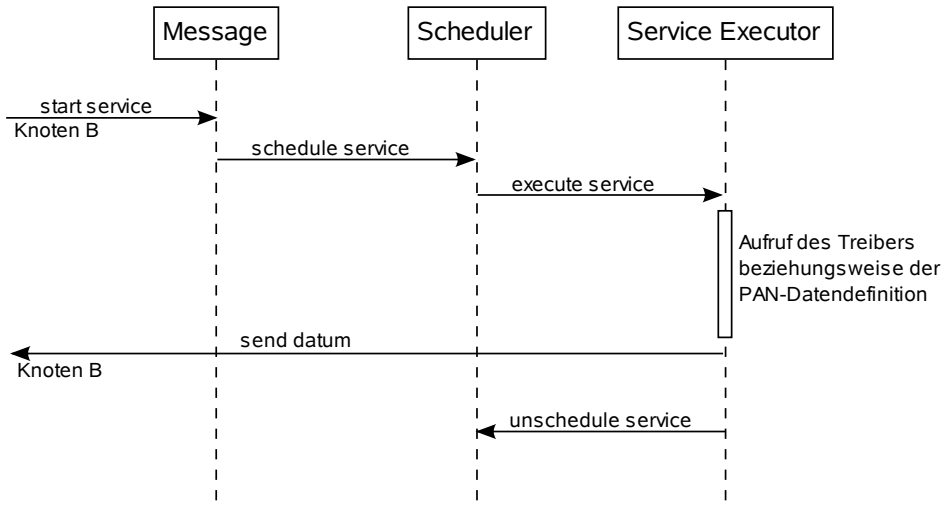

Abbildung 3.17: Ablauf der Ausführung eines *DataRead*-Diensts in der TalassaVM

In Abb. 3.17 ist dargestellt, wie *DataRead*-Dienste in der TalassaVM ausgeführt werden. Nach der Startaufforderung *start service* von Knoten B an die Komponente *Message* wird der Dienst im *Scheduler* über *schedule service* aufgenommen. Der *Scheduler* wiederum startet den Dienst im *Service executor* über *execute service*. Abhängig von der Dienstdefinition des entsprechenden Dienstes *DataRead* wird der zuständige Treiber oder die zuständige Pan-Datendefinition aufgerufen. Nach der Ausführung des Treibers oder der netzinternen Datenverarbeitung von Pan wird das erhaltene Sensordatum über *datum* an den Knoten B zurückgeschickt. Über *unschedule service* wird schließlich der Dienst aus dem *Scheduler* entfernt.

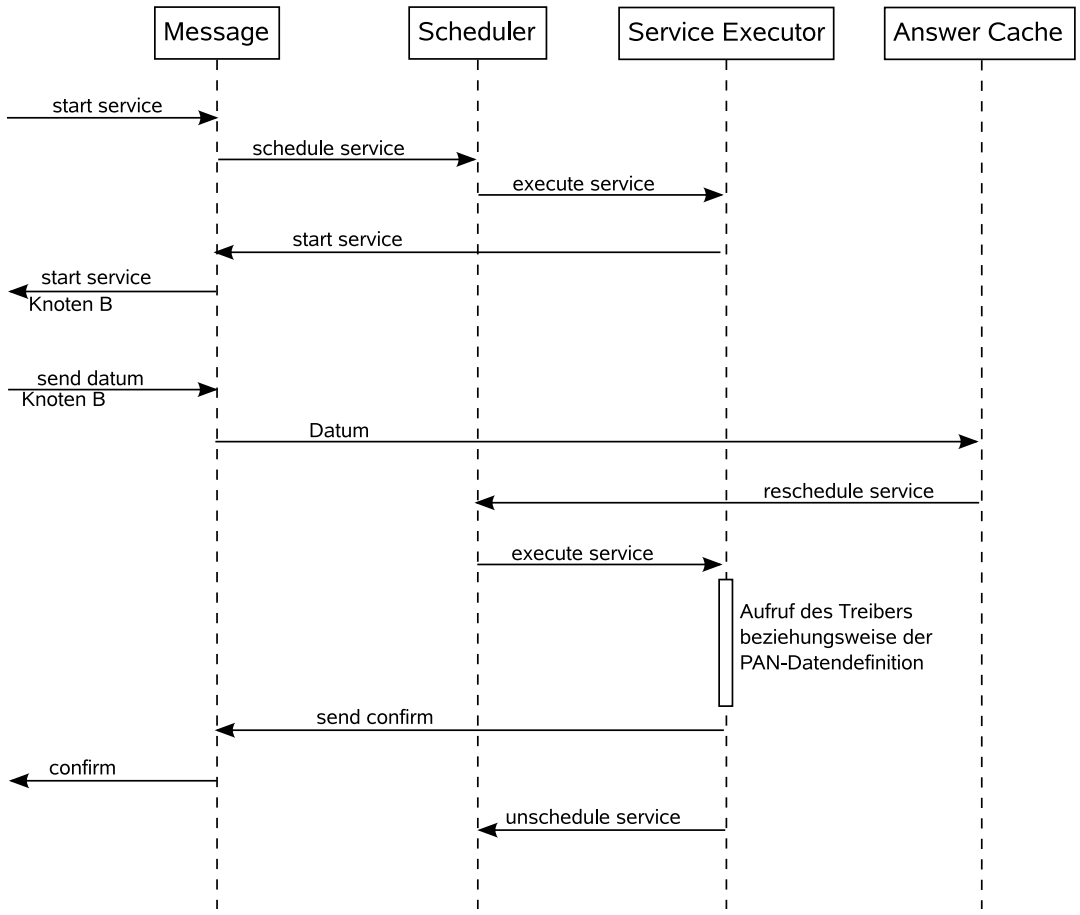

Abbildung 3.18: Ablauf der Ausführung eines *DataWrite*-Diensts in der TalassaVM

Ausführung von *DataWrite*

In Abb. 3.18 ist der Ablauf bei der Ausführung eines *DataWrite*-Dienstes dargestellt. Es wird hier angenommen, dass der Dienst *DataWrite* einen Wert schreibt, den er durch einen Dienst *DataRead* auf Knoten B erhält.

Nach der Startaufforderung durch *start service* an die Komponente *Message* wird der Dienst über *schedule service* im *Scheduler* aufgenommen. Dieser startet die Ausführung über *execute service* im *Service executor*. Der *Service executor* stellt fest, dass der entsprechende Parameter fehlt und startet deswegen den zum Parameter zugehörigen Dienst über *start service*. Die Komponente *Message* schickt diese Aufforderung mit *start service* an Knoten B.

Sobald das Datum von Knoten B bei der Komponente *Message* eintrifft, übergibt sie dieses an den *Answer cache*, der wiederum mit *reschedule service* den Dienst im *Scheduler* erneut einreiht. Nachdem der Dienst ausgeführt worden ist, wird dies mit *send confirm* bestätigt und der Dienst mit *unschedule service* aus dem *Scheduler* entfernt.

Ausführung von Ausführungslisten (*execution list*)

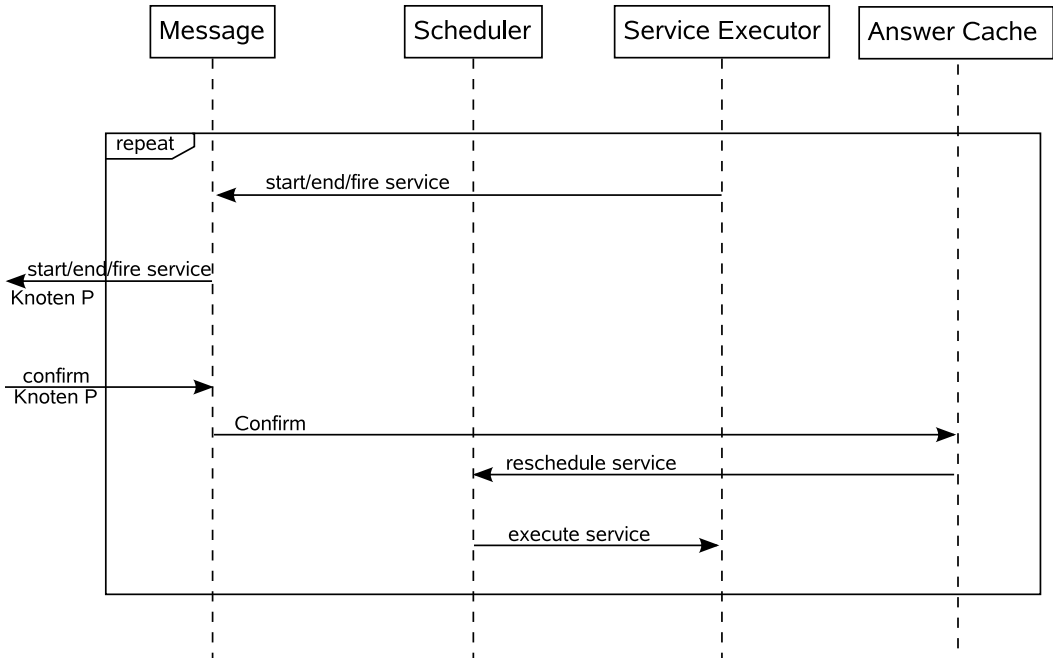

Abbildung 3.19: Ablauf der Ausführung von *execution list* in der TalassaVM

Sämtliche komponierende Diensttypen basieren auf Ausführungslisten (*execution list*). In Abb. 3.19 ist dargestellt, wie solche Ausführungslisten in der TalassaVM ausgeführt werden. Eine Ausführungsliste *Execution list* kann nach Definition aus folgenden Befehlen bestehen:

- *start service*,
- *stop service*,
- *fire event*.

Ausführungslisten treten in den komponierenden Diensten unter folgenden Namen auf:

- *Repetitive*: *execution list*

- *Conditional*: *else-list*, *then-list*
- *Event*: *event-list*, *timeout-list*

Der *Service executor* schickt den entsprechenden Befehl über die Komponente *Message* an den Knoten, auf dem der Dienst ausgeführt wird. In diesem Beispiel ist dies der Knoten P. Sobald der Knoten P den entsprechenden Befehl ausgeführt hat, bestätigt dieser über *send confirm* die Ausführung. Über den *Answer cache* und den *Scheduler* wird der *Service executor* erneut aufgerufen, und dieser löscht dann den Befehl aus der Liste der auszuführenden Befehle.

Der *Service executor* wiederholt dies solange, bis sämtliche Befehle der *execution list* ausgeführt sind.

Ausführung von *Repetitive*

Der Diensttyp *Repetitive* führt eine *execution list* wiederholt aus. Initial wird dieser Diensttyp wie alle anderen Diensttypen über *start service* gestartet. Bei den anschließenden wiederholten Ausführungen wird er über die Komponente *Timer* gestartet.

In Abb. 3.20 ist der Ablauf bei der initialen Ausführung dargestellt. Wie andere Dienste auch, wird der *Repetitive*-Dienst über *start service* der Komponente *Message* von einem Knoten P, über *schedule service* an den *Scheduler*, und über *execute service* an die Komponente *Service executor* gestartet. Der Dienst *Repetitive* startet dann über *start timer* einen *Timer*. Der Timer erhält den Parameter *interval* der Dienstbeschreibung des *Repetitive* übergeben. Anschließend wird die *execution list* einmalig ausgeführt. Im Anschluss an die einmalige Ausführung bestätigt der Dienst *Repetitive* seine erfolgreiche Ausführung über *send confirm* und wird über *unschedule service* aus dem *Scheduler* entfernt.

Der *Repetitive*-Dienst hat im Gegensatz zu den primitiven Diensten eine „lange“ Laufzeit. D. h. dieser Dienst wird nicht lediglich einmalig ausgeführt, sondern er wird über eine längere Laufzeit regelmäßig wiederholt. Potentiell kann dieser Dienst auch unbegrenzt oft wiederholt werden und hat dann eine unbegrenzte Laufzeit.

Nach dem Starten eines solchen Diensttyps (Abb. 3.20) wird die Ausführungsliste, von einem Timer gesteuert, wiederholt ausgeführt (Abb. 3.21). Sobald der *Timer* abläuft, stellt dieser den Dienst *Repetitive* erneut über *schedule service* in den *Scheduler*, der über *execute service* den Dienst erneut startet. Die Ausführungsliste *execution list* wird dann erneut ausgeführt.

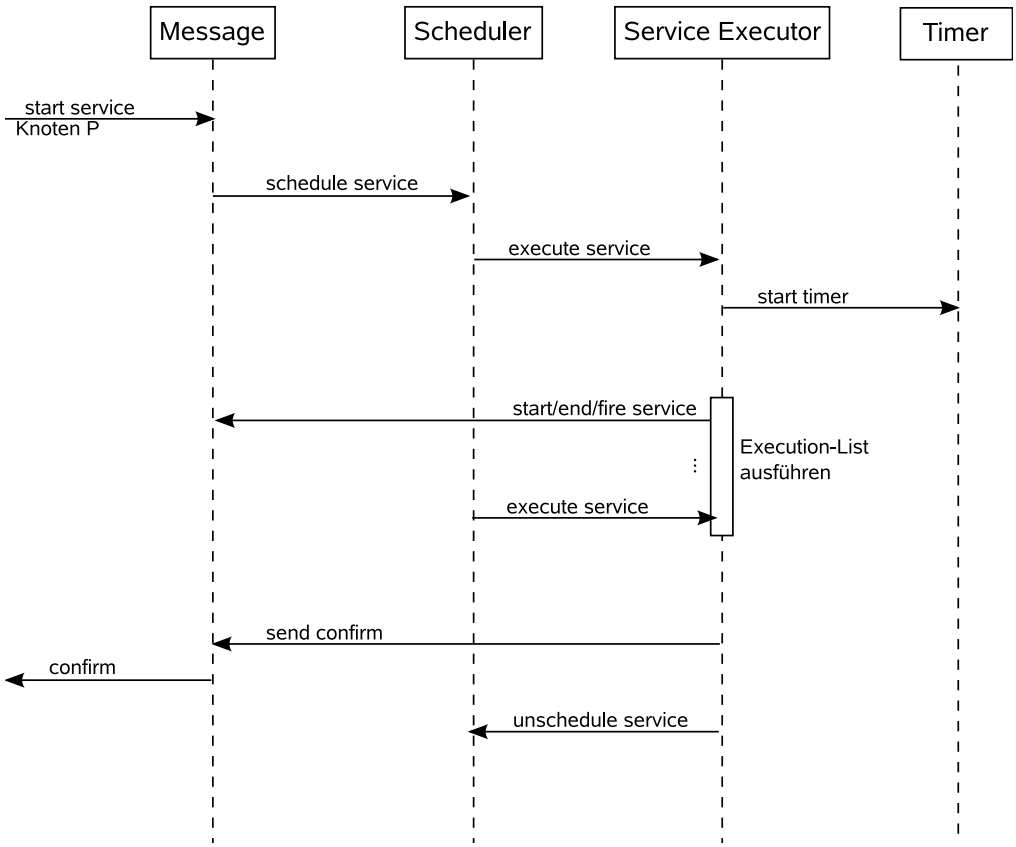

Abbildung 3.20: Ablauf der Ausführung eines *Repetitive*-Diensts in der TalassaVM: Startphase

Ausführung von *Conditional*

Die Ausführung eines *Conditional*-Dienstes ist in Abb. 3.22 dargestellt. Der Dienst *Conditional* wird über *stop service*, *schedule service* und *execute service* über die entsprechenden Komponenten *Message*, *Scheduler* und *Service executor* gestartet. Zuerst werden die in der Dienstbeschreibung definierten Parameter geholt (siehe dazu Abschnitt 3.5.1). Sobald alle Parameter vorhanden sind, wird die Bedingung der Dienstbeschreibung ausgewertet. Ist die Bedingung erfüllt, wird *then-list* ausgeführt, im anderen Falle *else-list*. Nach der Ausführung der entsprechenden *execution list* wird die Ausführung des Dienstes *Conditional* über *send confirm* bestätigt und der Dienst über *unschedule service* aus dem *Scheduler* entfernt.

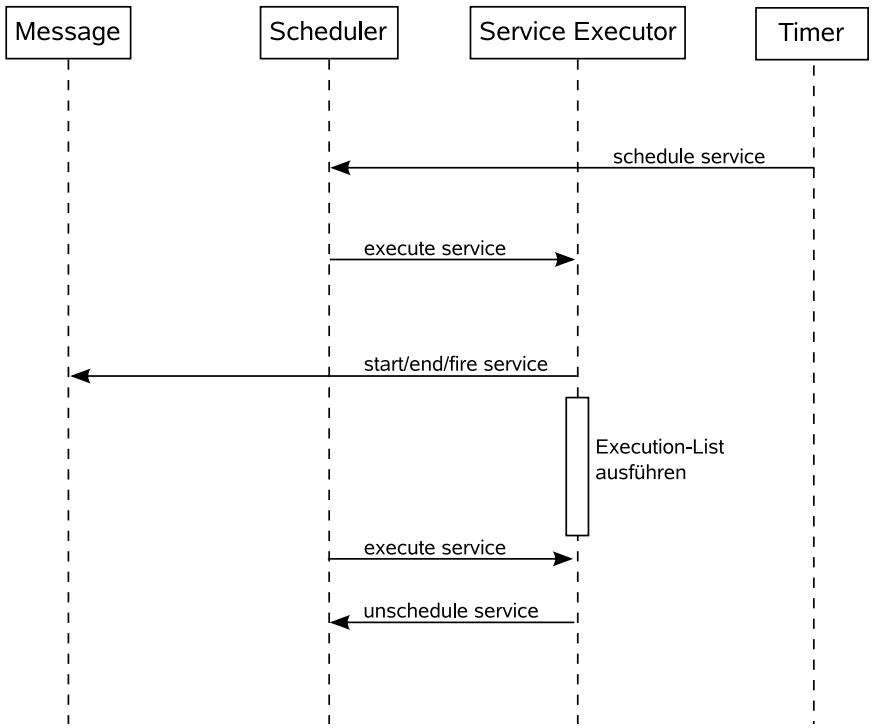

Abbildung 3.21: Ablauf der Ausführung eines *Repetitive*-Diensts in der TalassaVM: Ausführungsphase

Ausführung von *Event*

Der komponierende Diensttyp *Event* führt eine Ausführungsliste in Reaktion auf ein Ereignis aus, oder führt eine alternative Ausführungsliste nach einem Timeout aus. Der Ablauf der Startphase ist in Abb. 3.23 dargestellt. Der Dienst *Event* wird über *start service*, *schedule service* und *execute service* über die entsprechenden Komponenten *Message*, *Scheduler* und *Service executor* gestartet. Er startet dann über *start timer* auf der Komponente *Timer* einen Timer mit dem Parameter *timeout*.

In Abb. 3.24 ist der Ablauf dargestellt, wenn das zu diesem Dienst passende Ereignis auftritt. Sobald das zu diesem Dienst gehörende Ereignis über *fire event* empfangen wird, wird der Dienst über *schedule service* und *execute service* erneut gestartet und die entsprechende Ausführungsliste *event-list* ausgeführt. Der Dienst wird dann über *send confirm* bestätigt und über *unschedule service* aus dem *Scheduler* entfernt.

In Abb. 3.25 ist der Ablauf dargestellt, wenn das zu diesem Dienst passende Ereignis nicht auftritt. Sollte innerhalb der Timeout-Zeit *timeout* kein Ereignis auftreten, startet

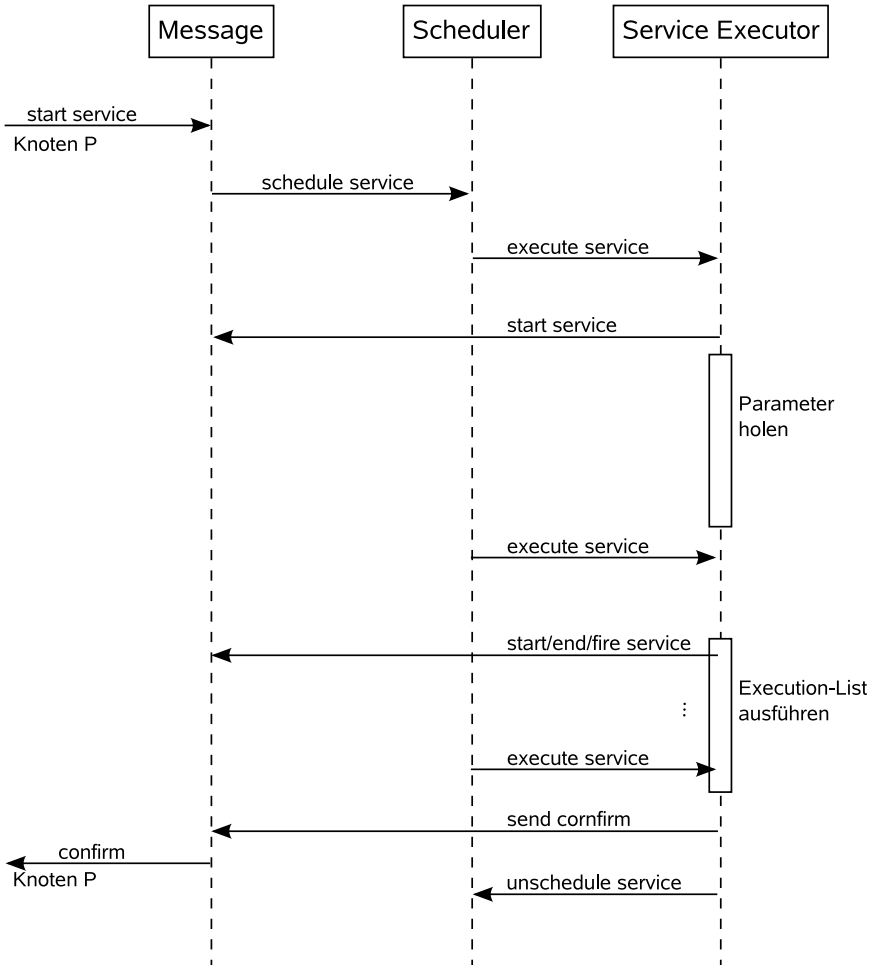

Abbildung 3.22: Ablauf der Ausführung eines *Conditional*-Diensts in der TalassaVM

die Komponente *Timer* über *reschedule service* den Dienst neu. In diesem Falle wird die alternative Ausführungsliste *timeout-list* ausgeführt. Der Dienst wird anschließend über *send confirm* bestätigt und über *unschedule service* aus dem *Scheduler* entfernt.

3.5.2 PanTalassaVM

In diesem Abschnitt wird die Integration der für die Pan-Daten zuständige Datenkomponente in die PanTalassaVM beschrieben. Wie in Abschnitt 3.4 beschrieben, kann die Kombination von Talassa mit Pan durch eine Ersetzung des Treiberzugriffs von *DataRead*-Diensten und *DataWrite*-Diensten durch den Zugriff auf die Pan-Datenbeschreibung erreicht werden. Um dies in der PanTalassaVM umzusetzen,

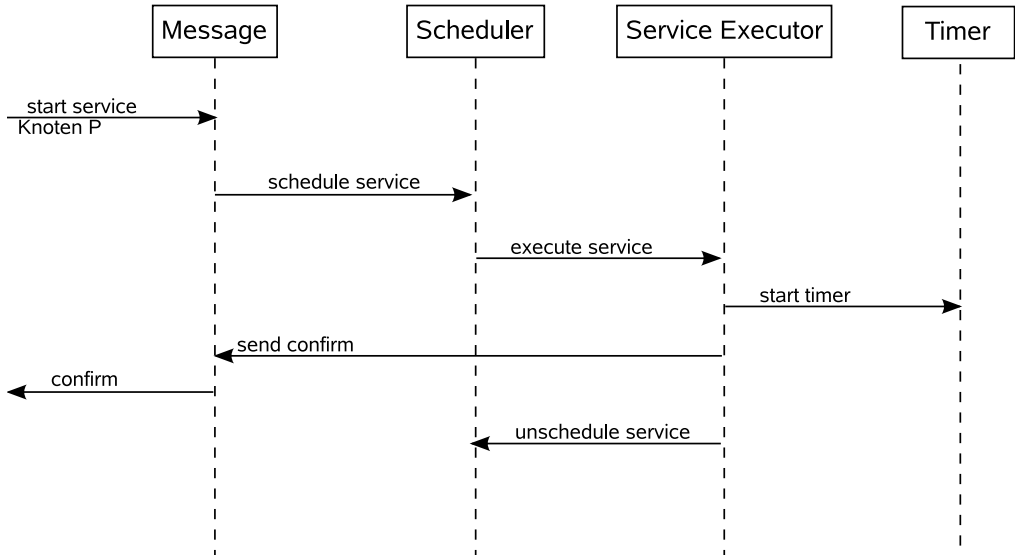

Abbildung 3.23: Ablauf der Ausführung eines *Event*-Diensts in der TalassaVM: Startphase

wird die Komponente *Answer cache* durch die Komponente *Data component* ersetzt. In Abb. 3.26 ist das ursprüngliche Komponentenmodell inklusive der Änderungen und Ergänzungen für die Integration von Pan dargestellt. Die in hellgrau dargestellten Komponenten sind die unveränderten Komponenten der TalassaVM. Die ursprüngliche Komponente *Answer cache* wird durch die schwarz markierte Komponente *Data component* ersetzt. Diese verfügt über die zusätzlichen Schnittstellen *read quantity* und *write quantity*. Über diese Schnittstellen kann die Komponente *DR executor* (Sensor-)Daten lesen und die Komponente *DW executor* Daten schreiben. Diese beiden Komponenten werden folglich dahingehend geändert, dass sie nicht direkt auf Treiber zugreifen, sondern stattdessen über die Komponente *Data cache* auf Sensoren, Aktoren oder bearbeitete Daten zugreifen. Die Komponente *Data component* besteht selbst aus mehreren Komponenten, die den Zugriff auf die Daten kontrollieren und die Bearbeitung der Daten ausführen. In Abb. 3.27 ist ein Überblick über alle Komponenten und ihre Beziehungen dargestellt.

I/O manager

Die Komponente *I/O manager* stellt nach außen die Schnittstelle *cache message* zur Verfügung und nutzt die Schnittstelle *reschedule service*. Diese Schnittstellen sind namens- und bedeutungsgleich zu den Schnittstellen der ersetzten Komponente *Answer cache*. Intern bietet die Komponente *I/O manager* die Schnittstellen *read data* und *write data*, mit denen sie den Zugriff auf physikalische Sensoren und Aktoren und den

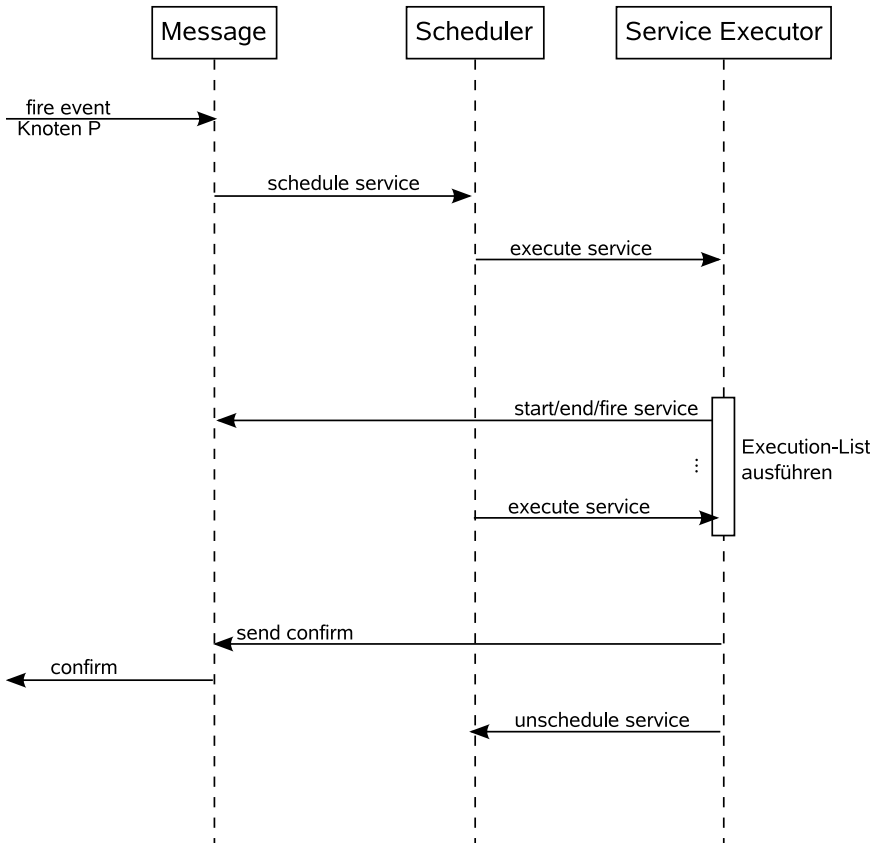

Abbildung 3.24: Ablauf der Ausführung eines *Event*-Diensts in der TalassaVM: „Fire“-Phase

internen Speicher erlaubt. Unterschieden werden die Zugriffe, wie in Abschnitt 3.3.3 beschrieben, anhand des Parameters *driver*.

Memory

Die Komponente *Memory* gewährt das Lesen und Schreiben von Daten auf dem knoteninternen Speicher. Der Parameter *driver* enthält in diesem Fall den Wert „Memory“.

Physical sensor/actuator

Die Komponenten *Physical sensor* und *Physical actuator* sind die Treiber zum Auslesen beziehungsweise Stellen des entsprechenden Sensors oder Aktors. Für jeden physikalischen Sensor und Aktor auf einem Sensorknoten existiert eine separate Komponente. Ist beispielsweise ein Temperatursensor auf dem Sensorknoten vorhanden, muss für diesen eine entsprechende Komponente implementiert werden, die den aktuellen Temperaturwert von dem physikalischen Temperatursensor ausliest und an die Komponente *I/O manager* weitergibt. Über den frei wählbaren Parameter *driver* wird dann der entsprechende Treiber referenziert. Im Falle des Temperatursensors kann als Parameter beispielsweise „TemperatureDriver“ gewählt werden.

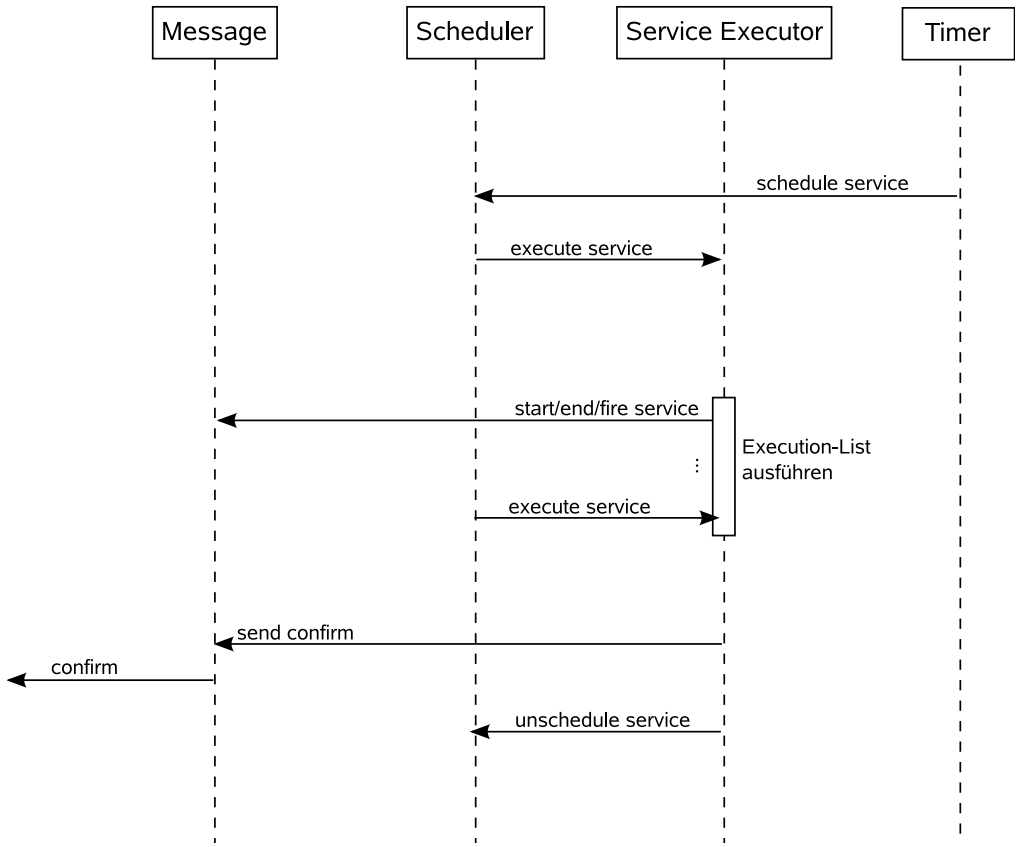

Abbildung 3.25: Ablauf der Ausführung eines *Event*-Diensts in der TALASSAVM: „Time out“-Phase

Data cache

Die Komponente *Data cache* verwaltet den flüchtigen Speicher, der sämtliche Nutzdaten zwischenspeichert, die von einem Sensorknoten empfangen werden. Bei wiederholter Anforderung desselben Datums kann somit die lokale Kopie verwendet werden, ohne dass ein erneutes Übertragen notwendig ist.

Data manager

Die Komponente *Data manager* ist die zentrale Stelle bei der Behandlung von Nutzdaten. Nach außen stellt sie über die Schnittstellen *read quantity* und *write quantity* den *DataRead*-Diensten und *DataWrite*-Diensten den Zugriff auf Nutzdaten zur Verfügung. Anhand der angegebenen Kennung *dataID* wird die dazugehörige Datenbeschreibung von der Komponente *Data description manager* geholt. Ist es eine Datenbeschreibung eines primitiven Datums werden die Rohdaten je nach Belegung des Parameters *driver* aus dem Speicher beziehungsweise von den physikalischen Sensoren geholt. Bei abgeleiteten Daten stammen die Rohdaten aus dem Cache. Die Rohdaten werden dann gemäß der Datenbeschreibung verarbeitet. Das verarbeite Datum wird dann dem aufrufenden

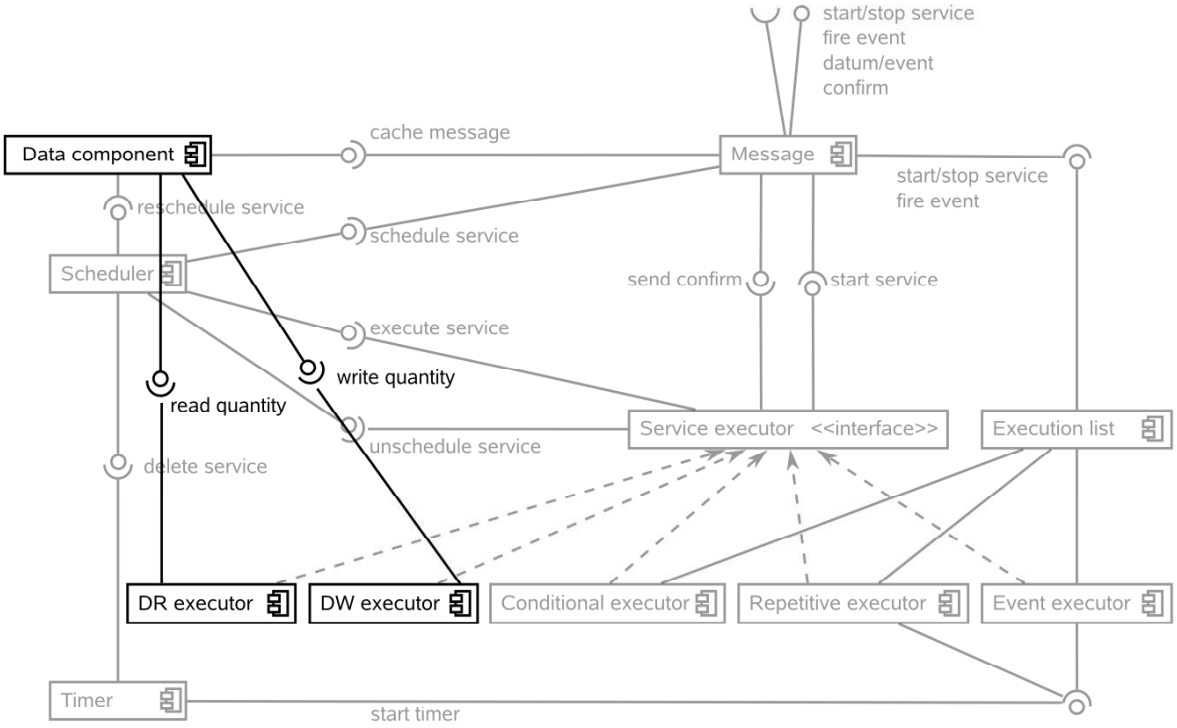

Abbildung 3.26: Vollständiges Komponentenmodell der PanTalassaVM, ergänzt mit Datenkomponente

DataRead-Dienst geliefert. Bei *DataWrite*-Diensten wird das Datum über die Komponente *I/O manager* dem zugehörigen Treiber übergeben.

Data description manager

Die Komponente *Data description manager* enthält die Datenbeschreibungen aller verwendeten Daten. Sie stellt diese der Komponente *Data manager* bei Bedarf zur Verfügung.

3.5.3 Auflösung der Dienst-Knoten-Zuordnung

Ein wichtige Aufgabe im dienstorientierten Sensornetz mit PanTalassa ist das Auffinden von Diensten. Dienste sind in PanTalassa prinzipiell deklarativ definiert. Dies bedeutet, dass u. a. der ausführende Knoten nicht von vornherein festgelegt werden muss. Wird ein Dienst mit einer bestimmten Dienstkennung *serviceID* gestartet oder gestoppt, oder wird ein Nutzdatum an einen anderen Dienst gesendet, muss deswegen derjenige Knoten gefunden werden, auf dem der Ziel-Dienst abläuft.

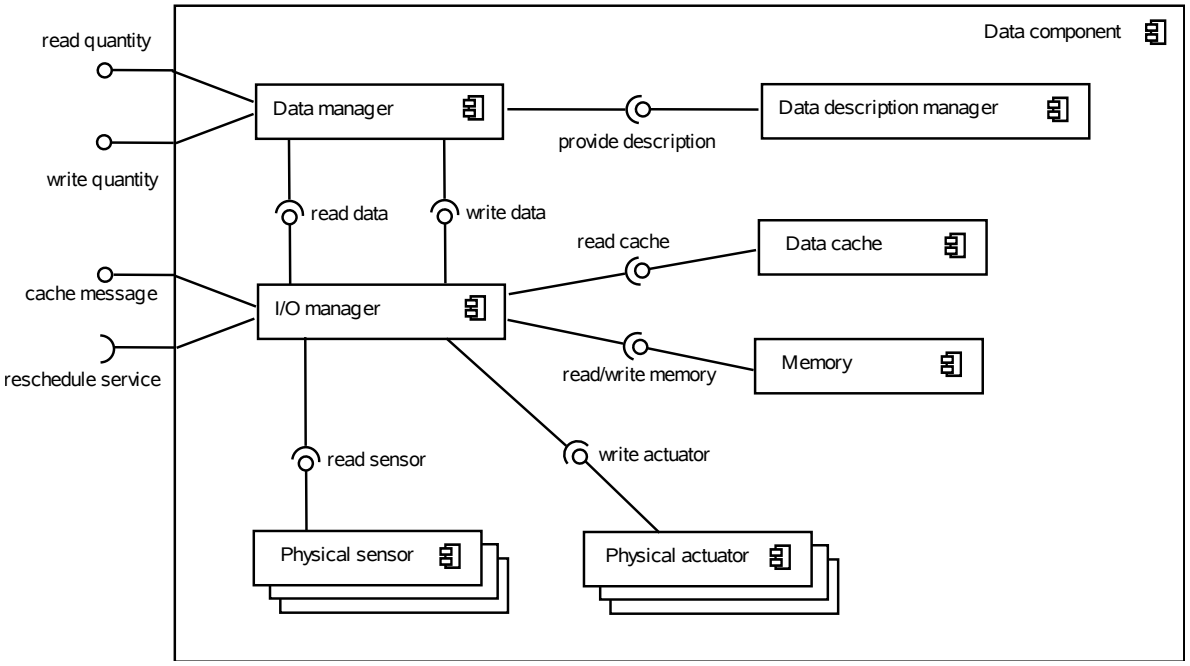

Abbildung 3.27: Komponentenmodell der Datenkomponente

Ein einfacher, effektiver Ansatz ist es, auf jedem Knoten eine komplette Liste mit der entsprechenden Zuordnung von Dienstkennung zu Knoten-Adresse anzulegen. Sobald ein Dienst auf einem Knoten einen anderen Dienst kontaktiert, wird diese Liste verwendet, um den betreffenden Knoten zu ermitteln. Dieser Ansatz ist allerdings nur dann effizient, wenn auf den Knoten genügend Speicherplatz für diese Liste vorhanden ist. Die Anzahl der Dienste darf somit nicht zu groß sein, damit die entsprechende Liste gespeichert werden kann. Ebenfalls ineffizient wird dieser Ansatz, wenn die Dienste den ausführenden Knoten häufig wechseln. Die Liste müsste dann auf allen Knoten aktualisiert werden.

Ein leicht modifizierter Ansatz ist die Speicherung der Zuordnungsliste auf einem dedizierten Sensorknoten. Dieser bietet als Verzeichnisdienst die Auflösung der Dienst-Knoten-Zuordnung, vergleichbar mit dem *domain name system* (DNS) des Internets (wobei der Knoten jedoch als einziger *name server* dient, ohne Delegierung, rekursiven Anfragen, etc.). Dieser Ansatz wird in der simulativen Evaluation von PanTalassa im Abschnitt 3.6.2 verwendet. Während das Problem der Änderung mit diesem Ansatz gelöst wird, da die Änderung immer nur auf einem Knoten erfolgt, bleibt jedoch das der Nicht-Skalierbarkeit bestehen.

Um das zentrale Speichern der Liste zu vermeiden, wurden dezentrale Systeme entwickelt, die eine effiziente Kommunikation in daten- beziehungsweise dienstorientierten Sensornetzen erlauben. Besonders geeignet sind dafür Ansätze, bei denen die vorhandenen Information von PanTalassa direkt zur Kommunikation genutzt werden können. Insbesondere ServiceCast ([KZ07], [Stu07]) und SCAN [Hof08] eignen sich für das von PanTalassa gewählte Modell mit Dienstkennungen und Dienstbeschreibungen, da sie sich im Gegensatz zu anderen Ansätzen nicht auf geographische Zonen (wie z. B. [VP05]) oder Informationen über die Nutzdaten (wie z. B. Directed Diffusion [IGE+03]) stützen. ServiceCast erlaubt eine direkte Kommunikation mithilfe von Dienstkennungen und einem Kontext, der die Funktionsweise des Dienstes umschreibt. SCAN ist ein sicherer Verzeichnisdienst, der eine verteilte Zuordnung von Dienstkennungen zu Knoten erlaubt.

Bei einer Umsetzung von PanTalassa für ein reales Sensornetz wurde SCAN als Verzeichnisdienst implementiert und verwendet. In Abschnitt 3.6.3 wird diese Umsetzung näher erläutert und evaluiert.

3.6 Evaluation

Im folgenden Kapitel wird der Einsatz von PanTalassa in realen Sensornetzen evaluiert. Besonderes Augenmerk wird dabei auf die Eignung der deklarativen Beschreibung bei der Erstellung, der Ausbringung und der Ausführung verteilter Sensornetzanwendungen gelegt.

Im ersten Teil werden Aspekte betrachtet, die für die Erstellung von verteilten Sensornetzanwendungen bedeutsam sind. Im Mittelpunkt stehen hier die inhärenten Eigenschaften des dienstorientierten Ansatzes von PanTalassa, die (unabhängig von der softwaretechnischen Umsetzung) die Erstellung und Ausbringung verteilter Sensornetzanwendungen unterstützen. Der Fokus ist hier, die Ausdrucksmächtigkeit und Flexibilität von PanTalassa als Programmierabstraktion qualitativ zu evaluieren, ohne bereits eine konkrete Umsetzung anzubieten oder quantitativ zu untersuchen.

qualitative Untersuchung

Die dienstorientierte Struktur von PanTalassa bietet jedoch nicht nur die Möglichkeit, verteilte Anwendungen zu programmieren, sondern lässt durch die deklarative Art der Beschreibung verschiedene Interpretationen bei der eigentlichen Ausführung im Sensornetz zu. Insbesondere die Auswahl der Sensorknoten, die einen Dienst ausführen, ist für die *Effektivität* der Anwendung unerheblich, bestimmt aber in entscheidendem Maße deren *Effizienz*. Im zweiten Teil werden deswegen Aspekte von PanTalassa betrachtet, die für die Ausführung einer verteilten Anwendung relevant sind, und die zu Effizienzsteigerungen genutzt werden können. Die angesprochenen Verfahren werden in einer simulativen Umsetzung quantitativ untersucht.

quantitative Untersuchung

Im dritten Teil wird die Implementierung und der Ablauf einer verteilten Anwendung in PanTalassa vorgestellt. PanTalassa wird hier genutzt, um ein intelligentes Gewächshaus in einem realen Sensornetz zu implementieren. Pflanzen werden hier durch entsprechende Dienste in einem lokalen Regelkreis automatisch bewässert und die Lichtverhältnisse werden, unter Einbezug der natürlichen Lichtverhältnisse und einem als Aktor dienendem künstlichem Pflanzenlicht, kooperativ so eingestellt, dass alle Pflanzen ihren Bedürfnissen gemäß mit Licht versorgt werden.

Einsatz im realen Sensornetz

3.6.1 Qualitative Beurteilung der Programmierung verteilter Sensornetzanwendungen mit PanTalassa

In den vorhergehenden Abschnitten wurde der Aufbau von Talassa-Diensten, Pan-Daten und deren Verwendung zur Implementierung von Sensornetzanwendungen beschrieben. Im Folgenden wird dargestellt, inwiefern die Eigenschaften der Dienst- und Daten-Definition die Implementierung und Ausführung von Sensornetzanwendungen unterstützen. Der Fokus liegt hierbei auf einer *qualitativen* Untersuchung und den *potentiellen* Möglichkeiten. Die vorgestellten Eigenschaften sind PanTalassa inhärent. In einer späteren *quantitativen* Untersuchung werden einige der hier vorgestellten Eigenschaften mit konkreten Umsetzungen verfeinert und quantitativ untersucht.

Deklarative Beschreibung der Anwendung

Mit Talassa-Dienstdefinition und Pan-Datendefinitionen wird eine Sensornetz-Anwendung deklarativ beschrieben.

Nutzdatenfluss

Zum einen gilt dies für das Lesen von (Nutz-)Daten auf Sensorknoten mit physikalischen oder virtuellen Sensoren, beziehungsweise das Schreiben von (Nutz-)Daten auf Sensorknoten mit physikalischen oder virtuellen Aktoren. Dieses Datenlesen und -schreiben mittels Sensoren und Aktoren wird durch die Definition primitiver Pan-Daten modelliert, die somit die „Ur“-Quellen und Senken des Nutzdatenflusses angeben. Des Weiteren wird durch Pan-Daten definiert, wie Nutzdaten weiterverarbeitet (z. B. aggregiert) werden können.

Datenverarbeitung

Kontrollfluss

Zum anderen definieren die komponierenden Dienste den Kontrollfluss zwischen Anwendungsteilen auf *Dienstebene*.

Deklarativ heißt somit in diesem Zusammenhang, dass nur die Datenerhebung, die Verarbeitung und der Kontrollfluss beschrieben werden. Dagegen wird nicht festgelegt, welche Knoten die jeweiligen Dienste ausführen sollen und wie der Kontrollverkehr zwischen diesen aussieht.

Dies ermöglicht eine flexible Auswahl der Zuordnung zwischen Diensten und Knoten. Der modellierte Kontrollfluss (Start und Stopp von Diensten, Auslösung von Ereignissen) ergibt den notwendigen Kontrollverkehr (Senden von Startnachrichten, Stoppnachrichten, Ereignisnachrichten) über das gewählte Schicht-3-Protokoll. Die Zu-

ordnung von Diensten auf Knoten bestimmt in Abhängigkeit von der Hopdistanz der Knoten den Aufwand für den Kontrollverkehr.

Ist es beispielsweise möglich, sämtliche Dienste auf einem einzigen Knoten ablaufen zu lassen, da dieser sowohl über sämtliche Sensoren wie auch über genügend Speicherplatz und Energie verfügt, wäre kein Kontrollverkehr über die Luftschnittstelle erforderlich. Wird davon ausgegangen, dass Kommunikation über die Luftschnittstelle mehr Energie verbraucht als Kommunikation innerhalb des Prozessors eines Sensorknotens, würde der über das gesamte Sensornetz ermittelte Energieverbrauch das theoretische Minimum erreichen.

Aber auch bei einer weniger extremen Verteilung der Dienste sind bei Anwendungen, die deklarativ identisch sind, in der konkreten Dienst-Knoten-Zuordnung Optimierungen möglich. Je kommunikationstopologisch näher kommunizierende Dienste sind, desto weniger Energie wird für den Kontrollverkehr benötigt.

Diese Eigenschaft von PanTalassa wird später im Abschnitt 3.6.2 quantitativ in Hinblick auf die Chancen zur Energieeinsparung untersucht.

Ressourcenbedarf und inhärente Parallelität

Die deklarative Definition einer verteilten Anwendung erfordert die Aufteilung der Programmlogik in die Bestandteile Datenfluss (*DataRead* und *DataWrite*) und Kontrollfluss (*Repetitive*, *Conditional* und *Event*). Jede einzelne Dienstdefinition besteht aus wenigen Dienstparametern (von sechs bei *DataRead* bis acht bei *Conditional*), d. h. sie benötigt sowohl beim Versenden im Sensornetz wie auch beim Speichern auf den Sensorknoten wenig Platz.

Gleichzeitig ist die Anwendung durch die Aufteilung in diese Speicher-schonenden Dienste implizit parallel implementiert: Solange keine Synchronisierung der Dienstausführung benötigt wird, können alle Dienste prinzipiell gleichzeitig ausgeführt werden. „Konventionelle“ prozedurale und objektorientierte Programmiersprachen wie Java oder C++ basieren auf dem Paradigma der implizit sequenziellen Ausführung der Befehle. Parallelität wird dagegen explizit durch Threads (oder Prozesse) erreicht. Alle Befehle innerhalb eines Threads werden wiederum sequentiell ausgeführt. Würde in einem Sensornetz ein einzelner Thread auf verschiedene Knoten verteilt, müsste die Sequenzialität durch zusätzliche Kommunikation zur Synchronisation erreicht werden.

Dienste sind dagegen implizit parallel. Jeder Dienst stellt einen „Thread“ dar. Im Vergleich zu „en-bloque“-Implementierungen stehen somit bei einer dienstorientierten Talassa-Anwendung mehr Threads zur Verfügung, die folglich eine echt parallele Ausführung auf unterschiedlichen Sensorknoten ermöglicht.

Die Dienstdefinitionen bieten jedem Sensorknoten die Möglichkeit, primitive Dienste bedingungslos sofort, und komponierende Dienst allein aufgrund der innewohnenden Parametervorgaben auszuführen. Eine in Talassa implementierte, verteilte Anwendung kann somit bezüglich der als atomar angenommenen Dienste die maximal erreichbare Parallelität erreichen, wenn alle Dienste vollständig auf unterschiedliche Sensorknoten verteilt sing.

Dies gilt insbesondere auch dann, wenn Dienste im laufenden Sensornetzbetrieb auf andere Knoten verschoben werden: Da die Kommunikation ausschließlich auf Dienstebene verläuft, müssen dienstausführende Knoten nicht auf netztopologische Zusammenhänge oder Änderungen Rücksicht nehmen (die Kommunikation auf Schicht 3 übernimmt das dort gewählte Protokoll).

Diese letztgenannte Eigenschaft der Ortsunabhängigkeit der Dienste ergibt sich auch aus den Ausführungen des vorangegangenen Abschnitts über die deklarative Natur von PanTalassa. Im Abschnitt 3.6.2 wird diese Eigenschaft dazu genutzt, Dienste energieoptimierend zu verschieben.

Verteilbarkeit von Diensten

Der letztgenannte Punkt der Möglichkeit zur energieoptimierenden Verschiebung legt das Augenmerk auf eine weitere inhärente Eigenschaft von PanTalassa-Anwendungen: freie Knotenzuordnung. Durch den weitgehenden Verzicht auf eine Festlegung des Ausführungsortes eines Dienstes steht es dem Programmierer frei, welche Sensorknoten mit der Ausführung eines Dienstes beauftragt werden:

Grundsätzlich kann jeder Knoten jeden *komponierenden* Dienst ausführen, und auch jeden *primitiven* Dienst, der auf virtuellen Sensoren/Aktoren basiert. Die einzige Einschränkung muss bei primitiven Diensten gemacht werden, die auf eine spezielle Hardware, beziehungsweise auf einen speziellen Ort angewiesen sind (z. B. ein *DataRead* zum Auslesen eines Lichtsensors am Fenster, oder ein *DataWrite*-Dienst zum Betreiben einer Jalousie am Fenster). Alle anderen Dienste einer verteilten Anwendung sind durch diese Eigenschaft frei auf Knoten zuordenbar.

Wenn die Sensorknoten dem Programmierer bekannt sind, kann diese Freiheit genutzt werden, um *a-priori* (vor der Ausbringung der Dienste) Sensorknoten festzulegen, die bestimmte Dienste ausführen sollen. Genauso kann aber auch *in-situ* (im laufenden Betrieb) die Knoten-Dienst-Zuordnung beliebig verändert werden, ohne dass die Funktionsfähigkeit der verteilten Anwendung beeinträchtigt wird (obgleich nicht-funktionale Eigenschaften wie Reaktionszeit, Energieverbrauch etc. durch die Veränderung der Zuordnung beeinflusst werden).

Besonders gefördert wird diese Flexibilität durch die zur Entwicklungszeit „erzwungene“ Aufteilung einer Anwendung in Dienste, was schließlich die Verteilbarkeit garantiert. Mit den diskutierten Einschränkungen bei primitiven Diensten kann es keine Dienste geben, die zwingendermaßen auf demselben Knoten ablaufen müssen. Deswegen kann jede Talassa-Anwendung durch die Aufteilung in Dienste verteilt ablaufen.

Die Granularität der Aufteilung wird durch die Auswahl der Basisdienste bestimmt. Im Falle von PanTalassa sind Datenerhebung, Bedingung, Wiederholung, usw. die Basisdienste und können nicht in einem monolithischen Dienst zusammengefasst werden. Da jeder Dienst potentiell auf einem anderen Sensorknoten ablaufen kann, kann jede mit PanTalassa geschriebene Anwendung in dieser Granularität flexibel verteilt und damit nach verschiedenen Gesichtspunkten optimiert werden.

Graphische Darstellung

Ein Vorteil der Beschreibung mit PanTalassa ist die Möglichkeit, verteilte Anwendungen graphisch darstellen zu können. Die in Abschnitt 3.2.6 eingeführte graphische Notation ermöglicht eine schnelle und übersichtliche Darstellung aller Dienste und deren Zusammenhang. Bei einer großen Anzahl von Diensten besteht die ebenso in Abschnitt 3.2.6 eingeführte Möglichkeit, Dienste zu Dienstmengen graphisch zusammenzufassen und somit auch große Anwendungen übersichtlich darzustellen. Durch diese Möglichkeit können verteilte Sensornetzanwendungen visualisiert und in ihrer Funktionsweise schnell erfasst werden. Dies dient nicht ausschließlich zur Veranschaulichung einer Anwendung, sondern nutzt insbesondere dem Verständnis für die Gesamtfunktion der Anwendung und zum Identifizieren fehlerhafter Bereiche.

Eine gesonderte Untersuchung der Eigenschaft „Graphische Darstellung“ ist nicht Teil dieser Arbeit. In Abschnitt 3.6.3 wird jedoch das Leitbeispiel „Intelligentes Gewächshaus“ vollständig graphisch modelliert.

Erstellen verteilter Anwendungen

Die Möglichkeit zur graphischen Darstellung kann jedoch nicht nur genutzt werden, um bestehende verteilte Anwendungen zu visualisieren, sondern auch, um verteilte Anwendungen mit graphischer Unterstützung zu entwickeln. Die Begrenzung auf fünf Diensttypen und auf wenige Parameter (maximal acht Parameter) für jeden Diensttyp ermöglicht eine fast komplett Quelltext-unabhängige Entwicklung verteilter Anwendungen. Dabei kann eine beliebige Abstufung der graphischen Unterstützung zur Anwendungsentwicklung gewählt werden. Durch den Einsatz von XML zur Dienstdefinition können beispielsweise standardisierte XML-Werkzeuge genutzt werden, um auf einfache Weise den XML-Quelltext zu generieren. Somit wird bei der Anwendung von XML-Schema-Definitionen (XSD) die syntaktische Korrektheit einer einzelnen Dienstdefinition garantiert.

Die Dienstorientierung bietet aber zusätzlich die Möglichkeit, spezialisierte Editoren zu entwerfen, die die speziellen Verknüpfungen zwischen Diensten (durch komponierende Dienste) unterstützen und somit nicht nur die syntaktische Korrektheit isolierter Dienstdefinitionen (wie bei XML mit XML-Schema-Definitionen), sondern auch eine dienstübergreifende syntaktische Korrektheit zu garantieren. Es kann dann überprüft werden, ob Dienste, die andere Dienste als Parameter beinhalten, auf tatsächlich vorhandene Dienste referenzieren.

Während die Erstellung eines Editors mit allen erwähnten Eigenschaften den Rahmen dieser Arbeit sprengen würde, wird in Abschnitt 3.6.3 ein Editor mit reduziertem Funktionsumfang kurz vorgestellt (siehe dazu Abb. 3.46, Seite 137).

Ausdrucksmächtigkeit und -effizienz von PanTalassa

Die Entwicklung von Anwendungen wird durch die in den Abschnitten „Graphische Notation“ und „Erstellen verteilter Anwendungen“ diskutierten Eigenschaften von PanTalassa vereinfacht. Dazu trägt insbesondere die Eigenschaft bei, dass bei PanTalassa die Anzahl der Diensttypen und deren Parameter auf eine kleine Anzahl reduziert ist (maximal acht Parameter), die die Ausdrucksmächtigkeit nicht verringert, aber gleichzeitig auf die Fähigkeiten kleiner Sensorknoten und die Anforderungen verteilter Sensornetzanwendungen zugeschnitten ist. Einfache verteilte Anwendungen, wie die regelmäßige Erhebung gleichartiger Sensordaten, die situationsbedingte Abfrage von Sensordaten und einfache Regelungssysteme können bereits mit einer geringen Anzahl von Diensten implementiert werden:

- Entwurfsmuster: Regelmäßige Erhebung gleichartiger Sensordaten
  - 1 *DataRead* (Auslesen des physikalischen Sensors)
  - 1 *Repetitive* (wiederholtes Starten des *DataRead*)
- Entwurfsmuster: Situationsbedingte Abfrage von Sensordaten
  - 1 *DataRead* (Auslesen des physikalischen Sensors)
  - 1 *Conditional* (Überprüfung der Situation/Bedingung und ggf. Starten des *DataRead*)
- Entwurfsmuster: Einfaches Regelsystem
  - 1 *DataRead* (Auslesen des physikalischen Sensors)
  - 1 *DataWrite* (Stellen des physikalischen Sensors)
  - 1 *Conditional* (Überprüfung, ob der gewünschte Wert dem tatsächlichen von *DataRead* entspricht; ggf. Starten des *DataWrite*, um den gewünschten Wert zu erreichen)
  - 1 *Repetitive* (wiederholtes Starten des *Conditional*)
    (siehe dazu auch die Modellierung der Helligkeitsregelung und Feuchtigkeitsregelung im intelligenten Gewächshaus in Abb. 3.43, Seite 133)

Oft wiederkehrende Muster in Sensornetzanwendungen können mit den aufgelisteten Entwurfsmustern durch wenige Dienste definiert werden und ermöglichen dadurch eine schnelle und korrekte Implementierung. Auch der Test gleichartiger Aufgaben, die sich durch verschiedene Parameterbelegungen unterscheiden, wird somit vereinfacht.

Dass durch die Beschränkung auf wenige, einfache Diensttypen die Ausdrucksmächtigkeit nicht eingeschränkt wird, konnte in Abschnitt 3.2.7 mit dem Beweis der Turingmächtigkeit nachgewiesen werden.

Abstraktion durch virtuelle Maschine PanTalassaVM

Der dienstorientierte Ansatz PanTalassa abstrahiert vom verwendeten Sensor-Betriebssystem und von der konkreten Hardware. Die virtuelle Maschine PanTalassaVM stellt aus architektureller Sicht diese Abstraktionsschicht dar. Sie ist auf der Basis des verwendeten Betriebssystems implementiert und stellt die Ausführung der einzelnen Diensttypen sicher. Die physikalischen Sensoren und Aktoren werden von der PanTalassaVM über Treiber angesprochen, die ebenfalls auf Basis des verwendeten Betriebssystems implementiert sind. Diese Treiber stellen somit die einzige Knoten- und

Hardware-nahe Implementierung dar, die zur Nutzung von Sensorknoten in PanTalassa-Anwendungen notwendig ist. Diese Abstraktion von Betriebssystem und Hardware (abgesehen von den notwendigen Treiberimplementierungen) erlaubt eine weitgehend Plattform-unabhängige Implementierung von verteilten Sensornetzanwendungen. Dies ermöglicht eine Nutzung heterogener Sensornetze, die aus Sensorknoten mit unterschiedlichen Plattformen und unterschiedlichen Betriebssystemen aufgebaut sind (die Kommunikationstechnik und das Schicht-3-Protokoll muss dagegen gleich sein, um Kommunikationsfähigkeit zu ermöglichen).

Ausbringung von Dienstdefinitionen ins Sensornetz

Die geringe Größe der einzelnen Dienstdefinitionen garantiert nicht nur eine effiziente Speicherung auf den Sensorknoten und eine einfache und schnelle Auswertung der Parameter zur Ausführung innerhalb der PanTalassaVM, sondern gewährt insbesondere ein effizientes Versenden der Dienstdefinitionen an die zugeordneten Sensorknoten. Ein Dienst wird durch seine Parameter vollständig beschrieben, weswegen zur Ausbringung einer Dienstdefinition auf einen Knoten das Versenden der Parameter ausreicht.

Als kompletter Quelltext in einer konventionellen Programmiersprache wie Java oder C++ hätten die Dienste einen wesentlich größeren Bedarf an Speicherplatz. Im Vergleich dazu können Talassa -Dienste schneller drahtlos im Sensornetz versendet und zur Ausführung gebracht werden.

Anpassbarkeit und Erweiterbarkeit verteilter Anwendungen durch PanTalassa

Die Aufteilung verteilter Sensornetzanwendungen in einzelne Dienste flexibilisiert nicht nur deren Ausführung und deren Anpassbarkeit während der Ausführung, sondern ermöglicht insbesondere auch die Änderung und die Erweiterung nach der Ausbringung. Laufende Anwendungen können durch den Austausch von Diensten im Nachhinein geändert beziehungsweise durch Hinzufügen von Diensten ergänzt werden. Dies schließt auch die Möglichkeit ein, neue Sensorknoten in ein bestehendes Netz nahtlos zu integrieren. Neue Sensorknoten könnten beispielsweise neue Dienste „mitbringen“, die sie in eine laufende Anwendung einfügen und damit sich selbst in die Anwendung integrieren.

Diese Eigenschaft, Anwendungen anpassen und erweitern zu können, wird in der Umsetzung des intelligenten Gewächshauses in einem realen Sensornetz genutzt, um neue Pflanzen im Gewächshaus zu integrieren. Neue Pflanzen bringen dort eine Reihe

von Diensten mit, die zu einer erweiterten und angepassten Regelung der Feuchtigkeit und Helligkeit führen (siehe hierzu Abschnitt 3.6.3).

Zusammenfassung der inhärenten Eigenschaften dienstorientierter Sensornetze mit PanTalassa

Die folgende Auflistung fasst die oben genannten Punkte mit den Eigenschaften von PanTalassa zusammen:

- inhärent parallele Ausführung verteilter Anwendungen
- inhärente Verteilbarkeit verteilter Anwendungen auf viele Sensorknoten
- Möglichkeit zur graphischen Darstellung und Erstellung verteilter Anwendungen
- wenig Ressourcenbedarf bei Speicherung und beim Versenden von Dienstdefinitionen
- ausdrucksmächtige Diensttypen (nachgewiesene Turingmächtigkeit)
- Abstraktion von Betriebssystem, Hardware und Plattform
- einfache Programmierung und Ausbringung verteilter Anwendungen „über die Luft“
- Erweiterbarkeit verteilter Anwendungen im laufenden Betrieb
- Anpassbarkeit verteilter Anwendungen im laufenden Betrieb

Diese Eigenschaften liegen im Aufbau von PanTalassa begründet und sind unabhängig von der konkreten Implementierung. Von den zahlreichen Möglichkeiten wird im Folgenden insbesondere der letztgenannte Punkt genauer untersucht und mit einem Verfahren konkretisiert und evaluiert.

3.6.2 Quantitative Untersuchung ausgewählter Optimierungsmöglichkeiten von PanTalassa

In diesem Abschnitt sollen die Möglichkeiten dargestellt werden, die der Einsatz von PanTalassa für die optimierte Ausführung der deklarativen Dienstdefinition bietet. Die freie Verteilbarkeit von Talassa-Diensten erlaubt eine nach gewissen Kriterien orientierte optimierte Knotenzuordnung. Für diese optimierte Knotenzuordnung wird im Folgenden ein konkretes Verfahren dargestellt und anschließend simulativ evaluiert.

Konkrete Knotenzuordnung

Jeder einzelne Dienst einer Dienstmenge muss, um seine Ausführung zu ermöglichen, einem Sensorknoten zugeordnet werden. D. h. seine Dienstbeschreibung muss auf einem Knoten gespeichert werden. Je nach Diensttyp ist diese Zuordnung mehr oder weniger frei möglich. Während die primitiven Diensttypen gewisse Randbedingungen bezüglich der Knotenzuordnung fordern, ist die Zuordnung der komponierenden Dienste abgesehen vom limitierten Speicherplatz vollständig frei. Bei den primitiven Diensttypen schränkt der notwendige Zugriff auf physikalische Sensoren beziehungsweise Aktoren die Auswahl bei der Knotenzuordnung ein. Zusätzlich ist bei vielen dieser Dienste auch der Ort des Sensorknotens entscheidend. Wird ein primitiver Dienst definiert, der die Helligkeit in einem bestimmten Raum ermitteln soll, kommen nur Sensorknoten in Frage, die einen Helligkeitssensor besitzen und dieser Sensor die Raumhelligkeit misst (und nicht beispielsweise die Außenhelligkeit vor dem Fenster).

Wesentlich freier gestaltet sich die Knotenzuordnung bei komponierenden Diensten. Dienste dieser Kategorie können auf beliebigen Sensorknoten ablaufen und können somit generell jedem Knoten zugeordnet werden. Lediglich die Speichergröße und die Verarbeitungskapazität eines Knotens begrenzen dabei die mögliche Anzahl der Dienste auf einem Knoten. Auch die durch Topologie und Routing begrenzte Erreichbarkeit anderer Knoten schränkt ggf. die freie Zuordnung ein: Wenn zwei Dienste miteinander kommunizieren, müssen diese solchen Knoten zugeordnet sein, die über das verwendete Routingprotokoll kommunzieren können.

Nutzung der Freiheitsgrade

Die deklarative Beschreibung einer Sensornetzanwendung durch eine mit Talassa definierte Dienstmenge und die dadurch gegebenen Freiheitsgrade bei der Knotenzuordnung können zu einer autonomen Zuordnung der Dienste vor und während des Ablaufs einer Anwendung genutzt werden.

In Abb. 3.28 ist eine beispielhafte Anwendung zur Erstellung eines aggregierten Sensorwertes dargestellt. Die mit Q bezeichneten Dienste stellen die Quellen dar, welche mit einem *DataRead*-Dienst einen Sensorwert liefern. Die mit A bezeichneten Dienste stellen aggregierende Dienste dar, welche die jeweiligen Quell-Sensorwerte zu einem aggregierten Wert zusammenfassen. Der mit S bezeichnete Dienst stellt den „letzten“ Aggregationspunkt dar, der über einen *DataWrite*-Dienst (welcher nicht explizit in der Abbildung dargestellt ist) den finalen aggregierten Wert an den Benutzer übergibt.

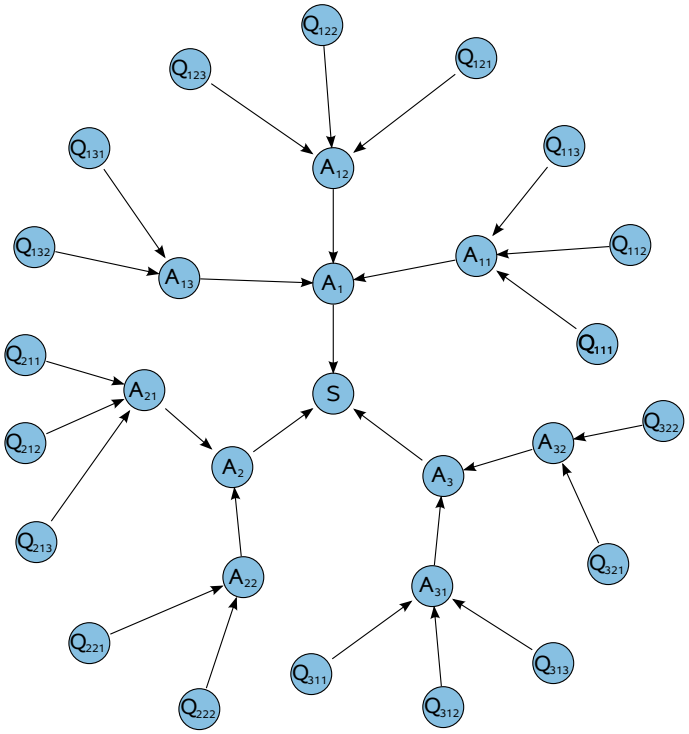

Abbildung 3.28: Kommunikationsbeziehungen der evaluierten Dienstmenge

Soll diese Dienstmenge (als deklarative Beschreibung einer Aggregation) auf ein reales Sensornetz abgebildet werden (d. h. die einzelnen Dienste werden individuellen Sensorknoten zugeordnet), sind nach obigen Ausführungen lediglich die primitiven Dienste zum Auslesen der Quellwerte, beziehungsweise zum Schreiben des finalen aggregierten Wertes ortsgebunden (d. h. einem bestimmten, geeigneten Sensorknoten zugeordnet). Je nach Lage der Quellen und der Senke können die frei zuordenbaren Aggregationsdienste günstigen Knoten zugeordnet werden.

Bei einem Sensornetz, in dem alle Knoten direkt und mit entfernungsunabhängigem Energieaufwand kommunizieren können, könnten dann sämtliche Aggregationsdienste dem Senke-Knoten zugeordnet werden, um den Kommunikationsaufwand zu minimieren. In der beispielhaften Anwendung in Abb. 3.28 würden dann alle aggregierenden Dienste A auf der Senke S liegen. Dies würde dazu führen, dass die 18 Quellen Q ihr Datum direkt an S senden. Liegen die Dienste A auf separaten Knoten sind dagegen 28 Kommunikationsnachrichten (18 Quelldaten und 10 Aggregationsdaten) notwendig. Ist keine direkte Kommunikation möglich oder steigen die Energiekosten mit der Entfernung, ist eine geänderte Zuordnung der Aggregationsdienste A erforderlich.

Optimierungsziele bei der Zuordnung

Die freie Zuordnung von Diensten kann für zahlreiche Ziele genutzt werden. Im vorhergehenden Abschnitt wurde der Kommunikationsaufwand beispielhaft als Kriterium genannt. Dieses und weitere Ziele werden in der anschließenden Liste aufgezählt und im Anschluss erläutert:

1. Kommunikationsaufwand (und -energie),
2. Kommunikationsgeschwindigkeit,
3. Kommunikationsrobustheit,
4. Lebensdauer,
5. Speicherplatz,
6. Sicherheit.

Wie schon im vorhergehenden Abschnitt erwähnt, können frei zuordenbare Dienste so zugeordnet werden, dass die Kommunikation wenig Energiekosten verursacht, indem diese Dienste in der Nähe ablaufen, oder sogar keine Energie braucht, wenn die Dienste auf demselben Knoten ablaufen (die Energie für die Kommunikation innerhalb eines Prozessors/Knotens wird hierbei als vernachlässigbar angesehen). Auf ähnliche Weise kann somit auch die Geschwindigkeit oder die Robustheit der Kommunikation optimiert werden. Notwendige Voraussetzung hierzu ist eine Metrik zur Einschätzung des Optimierungszieles sowie die Fähigkeit der einzelnen Knoten zur Auswertung der Metrik. Eine einfache Metrik für die drei eben erwähnten Ziele wäre die Hopdistanz zwischen kommunizierenden Diensten. Je geringer die Hopdistanz zwischen zwei kommunizierenden Diensten, desto geringer würde der Energieverbrauch eingeschätzt werden (und desto größer die Geschwindigkeit und Robustheit der Kommunikationsverbindung).

Natürlich ist diese Metrik im gleichen Maße einfach wie auch vereinfachend. In vielen Fällen stimmt die Annahme, dass kleinere Hopdistanzen auch geringerem Energieverbrauch entsprechen, nicht ohne weiteres. Es fließen in der Regel weitere Faktoren ein, wie z. B. der physikalische Abstand der Knoten, wenn die verwendete Funktechnik eine Regelung der Sendeleistung zulässt. Eine Kommmunikation über einen Zwischenknoten (d. h. zwei kurze Strecken) kann dann z. B. unter Umständen energieeffizienter sein, als eine Kommunikation ohne Zwischenknoten (d. h. eine lange Strecke). Auf eine kritische Behandlung solcher Phänomene und eine ausführliche Diskussion geeigneter Metriken soll hier aber verzichtet werden. Es genügt hier anzunehmen, dass solche Metriken exis-

tieren und die angesprochene Hopdistanz in grober Näherung eine geeignete Metrik darstellt.

Unter gewissen Voraussetzungen (nicht bewegliche Sensorknoten, gleichbleibende Kommunikationsbedingungen) sind die eben erwähnten Optimierungsziele statisch. Die Metrik bei Zielen dieser Art ändert sich nicht im laufenden Betrieb und die relevanten Messwerte (z. B. Hopdistanz) sind a-priori bekannt. Die Knotenzuordnung könnte somit ebenso a-priori festgelegt werden.

Es gibt aber auch Optimierungsziele, deren relevante Messwerte nicht notwendigerweise a-priori festgelegt sind, sondern sich erst im laufenden Betrieb eines Sensornetzes ergeben, beziehungsweise sich sogar währenddessen ändern. Zu einem solchen Optimierungsziel gehört beispielsweise die Lebensdauer eines Sensornetzes beziehungsweise einer Sensornetzanwendung. Der Energievorrat einzelner Sensorknoten ist meist nicht a-priori bekannt, da beispielsweise der Ladezustand von Batterien oder die Effektivität von Solarzellen ständigen Schwankungen unterworfen ist. In diesem Falle muss die Knotenzuordnung ad-hoc beim Start einer Anwendung vorgenommen werden und bei einer Änderung der relevanten Messwerte im laufenden Betrieb korrigiert werden.

Zu dieser Klasse an Optimierungszielen zählt auch der Speicherplatz. Werden beispielsweise (Nutz-)Daten von einzelnen Sensorknoten temporär gesammelt oder sogar permanent gespeichert, ändert beziehungsweise verringert sich der zur Verfügung stehende Speicherplatz für Dienste. Auch anhand des zur Verfügung stehenden Speicherplatzes kann die Zuordnung der Dienste ausgerichtet werden: Braucht ein Knoten mehr Speicher für andere Daten, können Dienste dieses Knotens im laufenden Betrieb anderen Knoten zugeordnet werden.

Neben Aspekten wie Energieverbrauch und Speicherkapazität können aber auch nicht-funktionale Ziele die Knotenzuordnung beeinflussen. Ein Beispiel eines nichtfunktionalen Optimierungsziels stellt Sicherheit dar. Unabhängig von anderen Optimierungszielen kann die Zuordnung eines Dienstes vom „Sicherheitszustand“ eines Knotens abhängen. Wird ein Knoten als gefährdet oder sogar korrumpiert angesehen, können Dienste anderen Knoten zugeordnet werden.

Simulative Untersuchung der Auswirkung situationsangepasster Knotenzuordnung

Im Folgenden wird die freie Knotenzuordnung nach verschiedenen Optimierungszielen beispielhaft gezeigt. Folgende Optimierungsziele werden dabei beachtet:

1. Kommunikationsaufwand,
2. Lebensdauer,
3. Speicheraufwand.

Jeder Knoten überwacht individuell und unabhängig von der Situation anderer Knoten seinen Speicherplatz und seine Energie. Dabei ist entscheidend, dass ein Knoten nur dann Dienste annimmt, wenn der zur Verfügung stehende Speicher und die zur Verfügung stehende Energie über einer festgelegten Schwelle liegen. Im Gegensatz zur rein Knoten-individuellen Überwachung von Lebensdauer (über Energie) und Speicheraufwand (über Speicherplatz) wird der Kommunikationsaufwand Knoten-übergreifend geschätzt. Zur Schätzung des Kommunikationsaufwandes bei der potentiellen Übernahme eines Dienstes dient die in Hops gemessene Distanz zum „Partner"-Dienst, also des Dienstes, der mit jenem kommuniziert.

Als Metrik, die die vorangegangen Optimierungsziele vereint, kann folgende Funkion dienen:

$$f(h, e, s) = min(g_h \frac{1}{h+1}; g_e e; g_s s) \tag{3.14}$$

Die Werte h, e, s geben hierbei die Hopdistanz h, die verbleibende Energie e und die verbleibende Speicherkapazität s an. Die Werte g_h, g_e, g_s sind die Gewichtungsfaktoren der entsprechenden Werte. Die Zuordnung eines Dienstes wird folgendermaßen vorgenommen: bei primitiven Diensten spielt die Hopdistanz keine Rolle, da sie als Quellen beziehungsweise Senken nicht frei zuordenbar sind. Komponierende Dienste werden denjenigen Knoten zugeordnet, deren Metrik f bezüglich dieses Dienstes maximal ist (d. h. je nach Gewichtung der Knoten mit der kleinsten Hopdistanz, der größten verbleibenden Energie oder des größten verbleibenden Speicherplatzes). Liegt ein Wert von 0 oder kleiner 0 vor, so verweigert der betreffende Knoten die Annahme eines weiteren Dienstes.

Wenn ein Dienst auf einem Knoten zugeordnet werden soll, werden alle Knoten nach der Auswertung dieser Funktion angefragt. Die Auswertung läuft auf den angefragten Knoten lokal ab und stellt eine Knoten-individuelle Einschätzung der Eignung für die Ausführung des Dienstes dar. Liegen dem anfragenden Knoten alle individuellen

Einschätzungen vor, kann er den Dienst dem geeignetsten Knoten zuordnen. Die Zuordnung kann abhängig von der Reihenfolge der Abfrage sein: Wenn ein Knoten bei der ersten Anfrage noch genügend Speicher hat und einen Dienst zugewiesen bekommt, kann dieser bei der Abfrage nach einem zweiten Dienst dessen Annahme verweigern, und der Dienst muss einem anderen Knoten zugeordnet werden.

Der Ablauf einer Knotenzuordnung soll an einem kurzen Beispiel erläutert werden. Die Gewichtungsfaktoren seien $g_h = 1000, g_e = 1, g_s = 1$. Ein Dienst a läuft auf einem Knoten x. Ein Dienst b hat eine Kommunikationsbeziehung zu Dienst a und soll einem Knoten zugeordnet werden. Im Sensornetz seien folgende drei Knoten mit den jeweiligen Parametern vorhanden:

- Knoten x
  - Hopdistanz zu Knoten x: $h = 0$
  - Energiezustand: $e = 400$
  - Freier Speicher: $s = 500$
- Knoten y
  - Hopdistanz zu Knoten x: $h = 1$
  - Energiezustand: $e = 800$
  - Freier Speicher: $s = 500$
- Knoten z
  - Hopdistanz zu Knoten x: $h = 2$
  - Energiezustand: $e = 700$
  - Freier Speicher: $s = 500$

Auf Anfrage des Knotens x liefern die drei Knoten folgende Antworten:

- Knoten x: $f_x = min(1000 \cdot 1; 1 \cdot 400; 1 \cdot 500) = 400$
- Knoten y: $f_y = min(1000 \cdot 1/2; 1 \cdot 800; 1 \cdot 500) = 500$
- Knoten z: $f_z = min(1000 \cdot 1/3; 1 \cdot 700; 1 \cdot 500) = 334$

Da Knoten y den größten Wert zurückliefert, wird der Dienst b diesem Knoten zugeordnet.

Vor dem Start einer Sensornetzanwendung müssen alle Dienste den Knoten im Sensornetz zugeordnet werden. Um dieses zu gewährleisten, werden in einer ersten Runde zuerst die Dienste zugeordnet, die auf einem festgelegten Knoten ablaufen müssen. Dies

sind alle primitiven Dienste, die einen festgelegten physikalischen Sensor beziehungsweise Aktor haben. Anschließend werden in einer zweiten Runde die Dienste verteilt, die eine direkte Kommunikationsbeziehung mit den Diensten der ersten Runde haben (z. B. ein *Conditional*-Dienst, der einen *DataRead*-Dienst als Parameter hat). Dazu fragen diejenigen Knoten, denen ein Dienst der ersten Runde zugeordnet ist, die anderen Knoten nach der Auswertung der Funktion f ab und ordnen den geeignetsten Knoten die Dienste der zweiten Runde zu. Für die dritte Runde werden die Dienste ausgewählt, die noch nicht zugeordnet sind und eine direkte Kommunikationsbeziehung mit Diensten der zweiten Runde haben (z. B. *Repetitive* startet *Conditional*). Auch hier fragt jeder betroffene Knoten der zweiten Runde die anderen Knoten nach der Auswertung der Funktion f ab und ordnet dem geeignetsten Knoten den Dienst der dritten Runde zu. Diese Runden werden analog solange wiederholt, bis alle Dienste zugeordnet sind.

Im laufenden Betrieb führen die Knoten die ihnen zugeordneten Dienste aus. Ein Dienst läuft auf einem Knoten, bis dieser eine Neuverteilung des Dienstes initiiert. Ein Knoten initiiert eine Neuverteilung, wenn sich seine lokalen Gegebenheiten so ändern, dass eine Weiterausführung nicht möglich oder sinnvoll ist. Eine Weiterausführung ist nicht mehr sinnvoll, wenn die Energie oder der Speicherplatz des Knotens unter eine zu definierende kritische Schwelle fällt:

$$g(e, s) = (e < e_{min} \vee s < s_{min}) \tag{3.15}$$

Sobald die Funktion g auf *wahr* evaluiert, initiiert der betroffene Knoten eine Neuverteilung seiner Dienste mit dem oben angegebenen Verfahren.

Alle Knoten reagieren während ihrer gesamten Lebenszeit auf Veränderungen der Bedingungen. Das bedeutet insbesondere, dass Knoten Dienste jederzeit zur Neuverteilung initiieren können, unabhängig davon, ob sie Dienste bereits „von Anfang an“ aufführen oder ob sie diese erst im Laufe der Zeit von anderen Knoten übernommen haben. Dienste können also im Laufe der Zeit mehrmals ihren Ausführungsort wechseln.

Im Folgenden wird das beschriebene Verfahren zur Knotenzuordnung quantitativ untersucht.

Simulationsparameter

Für die Funktion f werden als Gewichtungsfaktoren $g_h = 1000, g_e = 1, g_s = 1$ gewählt. Der Speicherplatz wird als abundant angenommen. D. h. den Knoten wird ein für alle Dienste ausreichender Speicherplatz zugewiesen. Ihren Energiezustand berechnen die

Knoten aus ihrer tatsächlich zur Verfügung stehenden Energie abzüglich einer Reserve von e_{min}. Für die Mindestenergie wird hierbei ein Wert von $e_{min} = 100$ gewählt. Die Gewichtungsfaktoren und die Mindestenergie wurden so gewählt, dass sie für die gewählten Simulationsparameter einen sinnvollen Ausgleich zwischen Hopdistanz und Energieverbrauch ergeben. Für anders geartete Simulationen oder reale Sensornetze müssen sinnvolle Werte geschätzt oder experimentell bestimmt werden.

Als Simulator wurde der Netzwerksimulator ns2 (siehe [ns2]) verwendet. Auf dessen Energiemodell basieren die Messungen des Energieverbrauchs. In diesem werden sämtliche Kommunikationsvorgänge proportional zur zeitlichen Dauer als Energieverbrauch gerechnet. In den Simulationen wird dem Standardmodell von ns2 folgend davon ausgegangen, dass das Senden um den Faktor 10 mehr Energie verbraucht als das Empfangen. Die *absoluten* Zahlen des Energieverbrauchs werden im ns2-Energiemodell nicht mit realen physikalischen Größen in Verbindung gebracht, sondern ermöglichen als abstrakte Energieeinheiten einen *relativen* Vergleich. Der Energieverbrauch beim Senden wird dabei mit $e_{send}(t) = 50 \cdot t$ berechnet, der Energieverbrauch beim Empfangen mit $e_{receive}(t) = 5 \cdot t$, wobei t jeweils für die benötigte Zeit für den Sende- beziehungsweise Empfangsvorgang steht.

AODV (siehe [PBRD03]) wird als eines der Standardprotokolle in Sensornetz-Simulationen verwendet. Auch in dieser Simulation wurde als Kommunikationsprotokoll AODV in der Standard-konformen Implementierung von ns2 gewählt. AODV wird sowohl zum Versenden der Dienstdefinitionen an die Knoten als auch zum Versenden des Kontrollverkehrs (Starten der Dienste) verwendet.

Das Sensornetz besteht insgesamt aus 100 Sensorknoten, die zufällig auf einem Areal von 1000 m x 1000 m verteilt sind. Die Sende- und Empfangsreichweite der Knoten wird auf 250 m festgelegt.

Ein zusätzlicher Master-Knoten übernimmt den Namensdienst für die Dienst-Knoten-Zuordnung. Bei diesem dedizierten Knoten werden die Informationen gesammelt, auf welchem Knoten ein Dienst abläuft, und er beantwortet Anfragen nach Dienst-Knoten-Zuordnungen. Dem Master-Knoten selbst werden keine Dienste zugeordnet.

Als Anwendung wird eine Dienstmenge verwendet, die Kommunikationsbeziehungen eines Aggregationsbaumes nachbildet. Sie besteht aus 40 *Repetitive*-Diensten. Jeder *Repetitive*-Dienst, der nicht ein Blatt des Baumes bildet, ruft regelmäßig drei weitere *Repetitive*-Dienste auf.

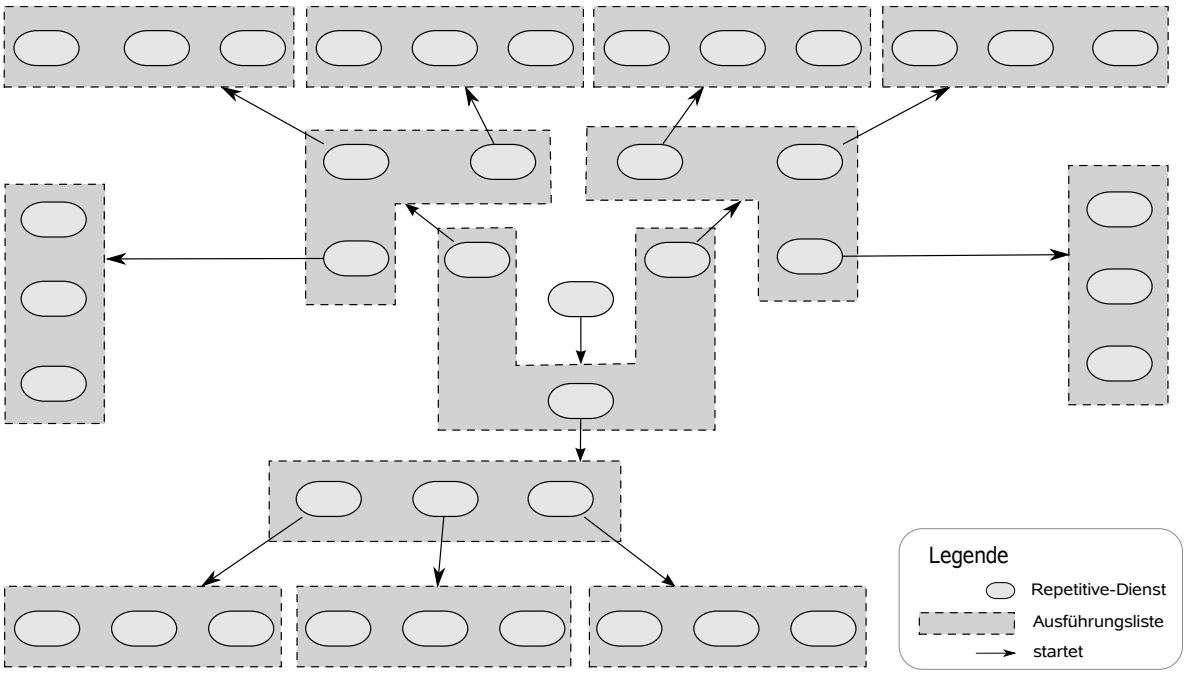

Abbildung 3.29: Aufbau und Kommunikationsbeziehungen der 40 *Repetitive*-Dienste in der Simulation

Die gesamte Simulationszeit teilt sich auf in die Initialisierungsphase, in der sämtliche Dienste den Sensorknoten zugeordnet werden, und die Ausführungsphase, in der die *Repetitive*-Dienste ausgeführt werden. Die Ausführungsphase ist auf 5000 s festgelegt (entspricht ca. 1,4 Stunden). Jeder der *Repetitive*-Dienste startet mit einer Periode von $interval = 15\,\text{s}$ drei andere Dienste, was 333 Startvorgängen für jeden der drei Dienste entspricht.

In Abb. 3.29 ist der von den *Repetitive*-Diensten gebildete ternäre Baum dargestellt. Hier sind auch die sich ergebenden Kommunikationsbeziehungen ersichtlich. Die Pfeile symbolisieren die Startnachricht zum Starten der Dienste in der Ausführungsliste. Der in der Abbildung zentrale Dienst startet in seiner Ausführungsliste drei weitere Dienste. Da er regelmäßig einen Startbefehl an diese Dienste schickt, ist die Kommunikationsbeziehung zu jedem dieser drei Dienste essentiell für das Funktionieren der Anwendung. Dies wird in der folgenden Evaluation insofern wichtig, da dort diese Kommunikationsbeziehungen verwendet werden, um die 40 Dienste energetisch günstig auf die 100 zur Verfügung stehenden Sensorknoten zuzuordnen.

In Tab. 3.18 sind die wichtigsten Simulationsparameter zusammengefasst. Eine vollständige Auflistung aller Parameter findet sich im Anhang in Abschnitt A.

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsprotokoll für Schicht 3	AODV
Kommunikationsreichweite	250 m
Speicherkapazität der Knoten	unendlich
Mindestenergie e_{min} der Knoten	100
Metrik f für Dienstverschiebung	$f = min(1000/(h+1); e; s)$
Metrik g für Dienstverschiebung	$g = (e < 100 \wedge s < \infty)$
Anzahl der Dienste	40
Intervall der Dienstaufrufe	15 Sekunden
Simulierte Zeit	5000 Sekunden ($\sim$ 1,4 Stunden)

Tabelle 3.18: Simulationsparameter zur Untersuchung der Verfahren zur Verschiebung der Dienste

Vergleichende Evaluation der situationsangepassten Knotenzuordnung

Im Folgenden werden drei Verfahren im gegenseitigen Vergleich evaluiert. Das erste Verfahren soll als Basisvariante ohne Optimierung den Vergleich zu den anderen beiden bieten. Dabei wird jeder der 40 Dienste in der Initialisierungsphase zufällig einem der 100 Sensorknoten zugeordnet. Die einmal ausgewählten Sensorknoten führen den ihnen zugeordneten Dienst bis zum Ende der Simulationszeit aus, ohne ihn auf andere Knoten zu verschieben. Im zweiten Verfahren werden die Knoten ebenfalls zufällig ausgewählt. Die Knoten können jedoch ihre Dienste während der Laufzeit auf andere Knoten verschieben. Im dritten Verfahren wird bereits in der Initialisierungsphase optimiert zugeordnet. Während der Laufzeit besteht hier ebenfalls die Möglichkeit, Dienste zu verschieben.

Die Initialisierung übernimmt ein fest definierter Master-Sensorknoten, auf dem zu Beginn der Initialisierung sämtliche Dienste gespeichert sind. Der Master-Knoten ordnet zuerst den in Abb. 3.29 zentralen *Repetitive*-Dienst einem anderen Knoten zu. Die restlichen Dienste ordnet er nach dem in 3.6.2 beschriebenen Initialisierungsverfahren den anderen Knoten zu.

Die folgende Aufzählung fasst die drei evaluierten Verfahren zusammen:

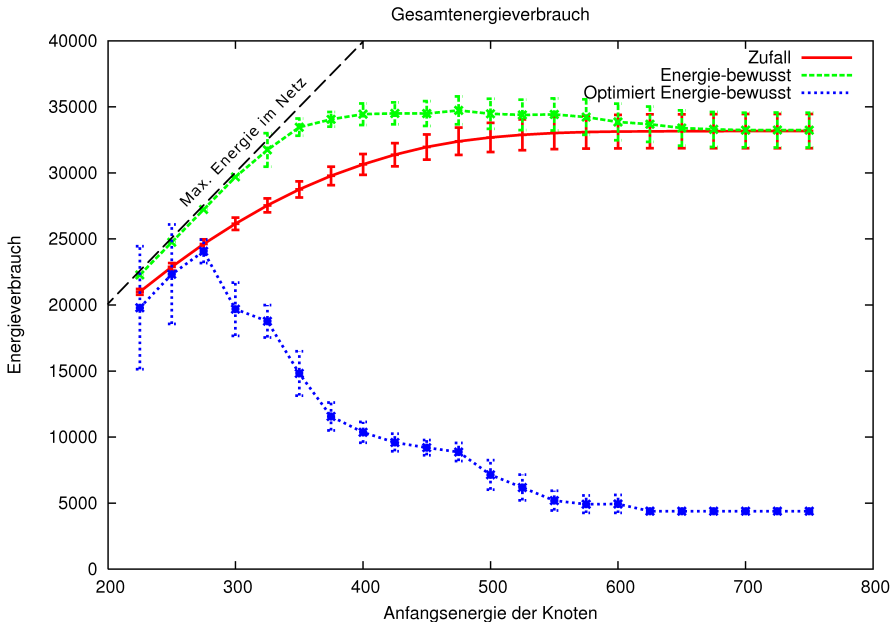

Abbildung 3.30: Kumulierter Energieverbrauch aller Sensorknoten (für die vollständigen Simulationsparameter siehe auch Tab. A.1 im Anhang)

- Zufall: Dienste werden zufällig zugeordnet und verbleiben während der ganzen Simulationszeit auf dem gewählten Knoten.
- Energie-bewusst: Dienste werden zufällig zugeordnet, wechseln aber bei über die Funktion g festgestellter Energieknappheit den Knoten. Dabei wird derjenige Knoten gewählt, der sich nach der oben genannten Funktion f als der geeignetste erweist.
- Optimiert Energie-bewusst: Dienste werden optimiert zugeordnet und wechseln bei über die Funktion g festgestellter Energieknappheit den Knoten. Als Metrik zur Zuordnung wird die oben aufgeführte Funktion f verwendet. Der Wechsel bei Energieknappheit folgt dem Schema des Szenarios „Energie-bewusst“.

Um die Unterschiede der Verfahren aufzuzeigen, werden deren Resultate bei unterschiedlichen Anfangszuständen der Energie untersucht. Jedem Knoten wird bei Simulationsbeginn die gleiche Anfangsenergie zugewiesen. Diese wird in den Simulationen von 220 bis 740 in Schritten von 20 variiert. Pro Stützpunkt (variierende Anfangsenergie) wurden 200 Simulationen durchgeführt.

In Abb. 3.30 ist auf der y-Achse der Energieverbrauch für die gesamte Ausführung der Anwendung zu sehen. Dazu zählen auch die Initialisierungsphase, in der die Diens-

te im Netz verteilt werden, und die eigentliche Ausführungsphase, in der die Dienste gemäß ihrer Definition ablaufen. Als Energieverbrauch wird hier der kumulierte Energieverbrauch aller Knoten gezählt. Insgesamt stehen dem Sensornetz mit 100 Knoten somit $e_{ges} = 100 \cdot e_{knoten}$ zur Verfügung. Diese gesamte zur Verfügung stehende Energie ist zur Orientierung als schwarze, lang-gestrichelte Gerade in Abb. 3.30 eingezeichnet. Sie beginnt bei (x = 200; y = 20.000) und verlässt bei (x = 400; y = 40.000) den dargestellten Koordinatenbereich.

Auf der x-Achse ist die Anfangsenergie der Knoten variiert. Mit genügend Anfangsenergie (ab 700 Energieeinheiten) bleiben alle Dienste über die gesamte Anwendung dem Knoten zugeordnet, dem sie in der Initialisierungsphase zugeordnet wurden, da keine Energieknappheit vorliegt. Energie steht den Sensorknoten in diesem Falle gleichsam abundant zur Verfügung. Der Energieverbrauch hängt deswegen alleine von der Anfangszuordnung ab. Deutlich zu sehen ist bei den Werten 700 bis 740 die Auswirkung der „optimiert Energie-bewussten“ Dienstzuordnung mit einem Gesamtenergieverbrauch von unter 5000 Energieeinheiten im Vergleich zu 32.000 Energieeinheiten bei zufälliger Zuordnung. Die Verfahren „Zufall“ und „Energie-bewusst“ unterscheiden sich bei diesen Anfangsenergiewerten nicht mehr, da bei beiden Verfahren die anfangs zufällige Dienstzuordnung ohne weitere Knotenwechsel beibehalten wird.

Je weniger Anfangsenergie die einzelnen Sensorknoten besitzen, desto häufiger ist ein Wechsel der Zuordnung im Laufe der Anwendung notwendig. Im Vergleich zur abundanten Energie, wo kein Dienst den Knoten wechselt, führt dies zu einem erhöhten Gesamtenergieverbrauch; sowohl beim Energie-bewussten Algorithmus als auch beim optimiert Energie-bewussten. Dieser Effekt wird desto stärker, je geringer die Anfangsenergie der Knoten ist.

Steht sehr wenig Energie zur Verfügung, wird annähernd die gesamte im Sensornetz zur Verfügung stehende Energie verbraucht. Dies ist besonders bei der geringsten simulierten Anfangsenergie von 220 Energieeinheiten augenfällig. Hier wird bei allen drei Verfahren fast die gesamte Energie (22.000 Energieeinheiten) verbraucht. Dies deutet bereits darauf hin, dass alle Knoten am Simulationsende keine Energie mehr haben, d. h. ausgefallen sind. Dies wird später noch genauer untersucht. Mit steigender Anfangsenergie steigt dann der Gesamtverbrauch analog zur höheren Gesamtenergie an. Speziell das „Energie-bewusste“ Verfahren nutzt die Energie bis zu einer Anfangsenergie von 300 vollständig aus, da bei Energieknappheit sofort der Dienst auf einen anderen Knoten wechselt. Erst danach wird nicht mehr die gesamte Energie des Sensornetzes verbraucht, da für die Wechsel und die Dienstausführung mehr Energie zur Verfügung steht. Trotzdem steigt der Energiebedarf noch bis ca. x = 400 an, da noch bei vielen Kno-

ten während der Simulationszeit die Energie unter die kritische Marke von $e_{min} = 100$ fällt, was einen Wechsel auslöst. Erst ab einer Anfangsenergie von etwa 400 wird das Maximum erreicht, um dann langsam bis zum Erreichen der abundanten Energie (bei Anfangsenergie 700) zu sinken.

Beim Verfahren „Zufall“ steigt der Energieverbrauch stetig bis zum Erreichen des Maximums bei einer Anfangsenergie von ca. 600. Da hier die Anfangszuordnung erhalten bleibt, wird stets soviel Energie verbraucht, wie es die Konstellation der Dienste zulässt. Knoten mit Diensten und Knoten auf den Kommunikationsrouten verbrauchen viel Energie, bei Energieknappheit sogar ihre vollständig Energie. Knoten, die nicht auf den Kommunikationsrouten liegen, verbrauchen dagegen sehr wenig Energie. Deswegen wird beim Verfahren „Zufall“ nie der gesamte Energievorrat des Netzes ausgeschöpft.

Beim „optimiert Energie-bewussten Verfahren“ sinkt der Energieverbrauch bereits ab einer Anfangsenergie von 280, da die optimierte Dienst-Knoten-Zuordnung zu einem wesentlich geringeren Energieverbrauch und weniger Wechseln führt. Der Energieverbrauch sinkt durch die stetig weniger werdenden notwendigen Wechsel, bis bei etwa 620 dessen Minimum erreicht wird. Ab diesem Wert (Anfangsenergie 620) ist demnach soviel Energie vorhanden, dass keine Wechsel mehr notwendig sind. Das „optimiert Energie-bewusste“ Verfahren verbraucht hier nur etwa 15 % der Energie im Vergleich zum Verfahren „Zufall“.

Im Folgenden werden die drei Verfahren bezüglich des Aufwandes während der Initialisierungsphase und der Ausführungsphase verglichen. Dazu wird die Gesamtzahl der durch das Schicht-3-Protokoll ausgelösten MAC-Pakete gezählt, die während der entsprechenden Phase von einem Knoten verschickt werden, dem ein Dienst zugeordnet ist.

In der Initialisierungsphase werden die Dienste den Knoten zugeordnet. In Abb. 3.31 ist die Summe der MAC-Pakete zu sehen, die während der Initialisierungsphase versendet werden. Dazu zählt bei allen drei Verfahren die Verteilung der Dienste im Netz und beim „optimiert Energie-bewussten“ Verfahren die Ermittlung der geeigneten Knoten. Unabhängig von der zur Verfügung stehenden Anfangsenergie ist die Anzahl der gesendeten MAC-Pakete bei jedem Verfahren für sich genommen gleich. Insbesondere ist die Anzahl beim „zufälligen“ und „Energie-bewussten“ Verfahren gleich, da diese dieselbe zufällige Initialisierungsstrategie verfolgen. Beim „optimiert Energie-bewussten“ Verfahren ist die Anzahl im Vergleich um etwa den Faktor 2 kleiner. Dies rührt daher, dass die Dienste bevorzugt denjenigen Knoten zugeordnet werden, die sich in kleiner Hopdistanz zum initialisierenden Masterknoten befinden. Dadurch werden in erhebli-

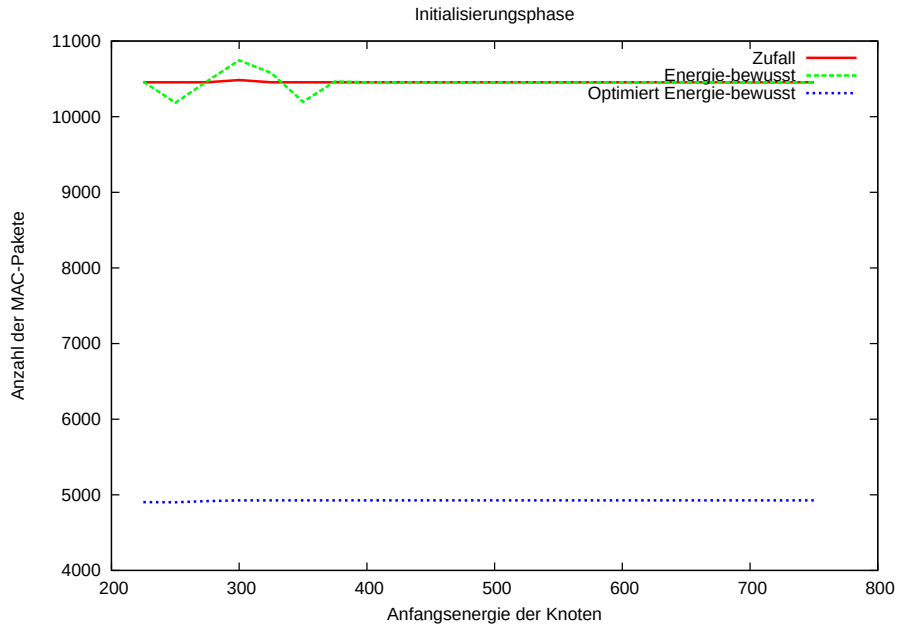

Abbildung 3.31: Versendete MAC-Pakete in der Initialisierungsphase (für die vollständigen Simulationsparameter siehe auch Tab. A.1 im Anhang)

chem Maße Kommunikationskosten eingespart, die ansonsten durch ein Weiterleiten über mehrerer Knoten hinweg entstünden.

In Abb. 3.32 ist die Anzahl der MAC-Pakete zu sehen, die während der Ausführungsphase der Anwendung versendet werden. In der Ausführungsphase versendet das „zufällige“ Verfahren unabhängig von der zur Verfügung stehenden Anfangsenergie stets gleich viele MAC-Pakete, da keine Dienste den Knoten wechseln und deswegen die Anzahl ausschließlich von der stets gleichbleibenden Dienstausführung herrührt. Bei den beiden anderen Verfahren ist die Anzahl der versendeten MAC-Pakete jedoch zusätzlich abhängig von den gewechselten Knotenzuordnungen. Somit werden bei geringerer Anfangsenergie mehr MAC-Pakete versendet als bei höherer Anfangsenergie, bei der weniger Zuordnungswechsel notwendig sind. Wegen des insgesamt höheren Energieverbrauchs beim „Energie-bewussten“ Verfahren sind die Zuordnungswechsel häufiger und damit im Vergleich zum „optimiert Energie-bewussten“ Verfahren die versendeten MAC-Pakete häufiger.

Eine wichtige Rolle für die Einschätzung der Funktionsfähigkeit eines Sensornetzes stellt die Anzahl der ausfallenden Knoten dar. In der Simulation werden die Knoten gezählt, die am Ende der Simulationszeit ausgefallen sind. Als ausgefallen zählt ein

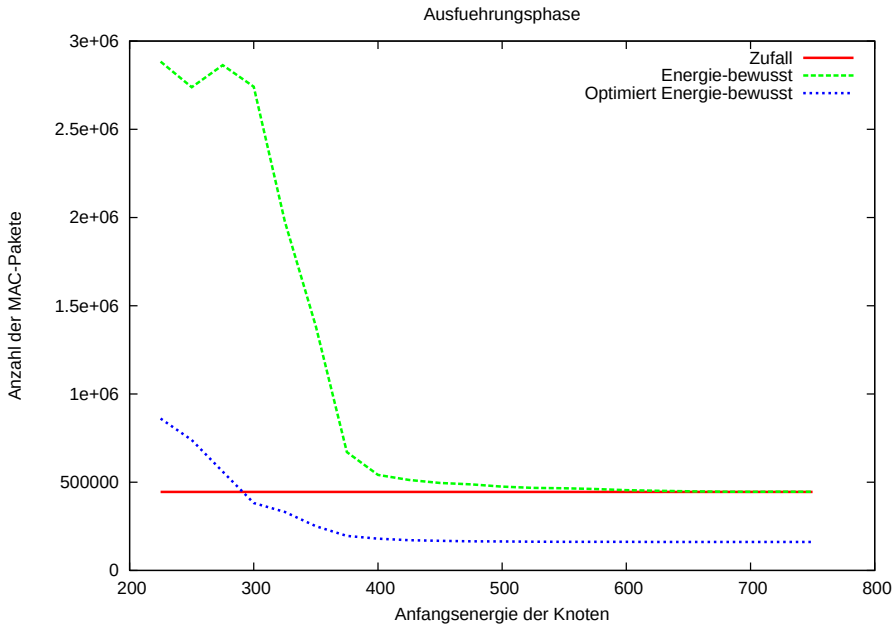

Abbildung 3.32: Versendete MAC-Pakete in der Ausführungsphase (für die vollständigen Simulationsparameter siehe auch Tab. A.1 im Anhang)

Knoten dann, wenn seine Restenergie den Wert 0 erreicht hat. Ob ausgefallene Knoten tatsächlich die Funktionsfähigkeit beeinträchtigen hängt davon ab, ob auf diesem Knoten Dienste ablaufen oder ob er zur Kommunikation benötigt wird. Auf diese Frage wird später noch eingegangen.

In Abb. 3.33 wird die Anzahl der ausgefallenen Knoten bei den drei Verfahren verglichen. Durch die notwendigen Zuordnungswechsel fallen bei einer geringeren Anfangsenergie sowohl beim „Energie-bewussten“ als auch beim „optimiert Energie-bewussten“ Verfahren zum Teil deutlich mehr Knoten aus als bei einer rein zufälligen Zuordnung. Während beim „optimiert Energie-bewussten“ Verfahren der Mehrbedarf rasch durch die bessere Anordnung amortisiert wird, fallen beim „Energie-bewussten“ Verfahren, bis bei etwa 700 Energieeinheiten Anfangsenergie, stets mehr Knoten aus als beim „zufälligen“ Verfahren. Ab dieser Anfangsenergie fallen schließlich keine Knoten mehr aus.

Während die Anzahl der insgesamt ausgefallenen Knoten einen groben Überblick über die Funktionsfähigkeit des Sensornetzes gibt, ist für den korrekten Ablauf einer verteilten Anwendung die wichtigste Kennzahl, wie viele der Dienste am Ende noch „funktionieren“. D. h. in diesem Falle, wie viele Dienste solchen Knoten zugeordnet sind, die zum einen noch genügend Energie zur Ausführung besitzen und zum anderen mit den

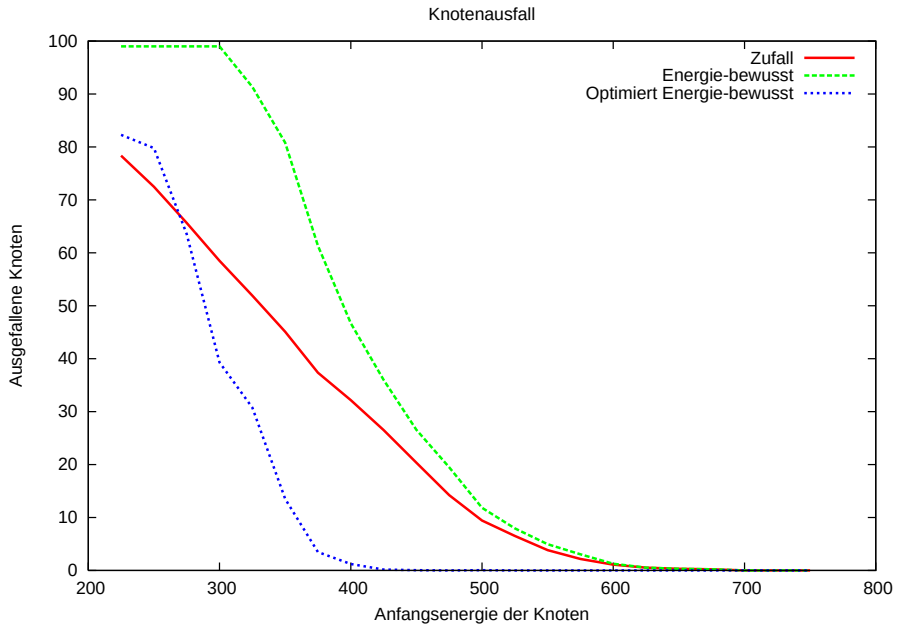

Abbildung 3.33: Ausgefallene Sensorknoten am Ende (für die vollständigen Simulationsparameter siehe auch Tab. A.1 im Anhang)

verbundenen Diensten noch kommunizieren können. Bereits der Ausfall eines Dienstes kann hier unter Umständen die Effektivität einer verteilten Anwendung stören oder völlig verhindern.

In Abb. 3.34 wird die Anzahl der ausgefallenen Dienste bei den verschiedenen Verfahren verglichen. Hier zeigt sich ein deutlicher Unterschied. Bei zufälliger Zuordnung ohne Wechsel fällt ab 700 Energieeinheiten Anfangsenergie kein Dienst mehr aus. Beim „Energie-bewussten“ Verfahren liegt dieser Punkt dagegen schon bei etwa 420 Energieeinheiten, obwohl sowohl der Gesamtenergieverbrauch als auch die Anzahl der ausgefallenen Knoten stets wesentlich höher ist als beim „zufälligen“ Verfahren. Durch den geringeren Energieverbrauch beim „optimiert Energie-bewussten“ Verfahren liegt bei diesem die Grenze, bei der keine Dienste mehr ausfallen, noch etwas niedriger. Ab 360 Energieeinheiten Anfangsenergie fällt hier kein Dienst mehr aus.

Wenn davon ausgegangen wird, dass sämtliche Dienste für eine Anwendung gebraucht werden, zeigt sich hier, dass die Lebenszeit der Anwendung mit dem „optimiert Energie-bewussten“ Verfahren deutlich verlängert wird. Eine Anwendung wird hierbei als lebend, d. h. funktionsfähig, bezeichnet, wenn sich alle ihrer Dienste auf nicht-ausgefallenen Knoten befinden und diese Dienste auch miteinander kommunizieren können.

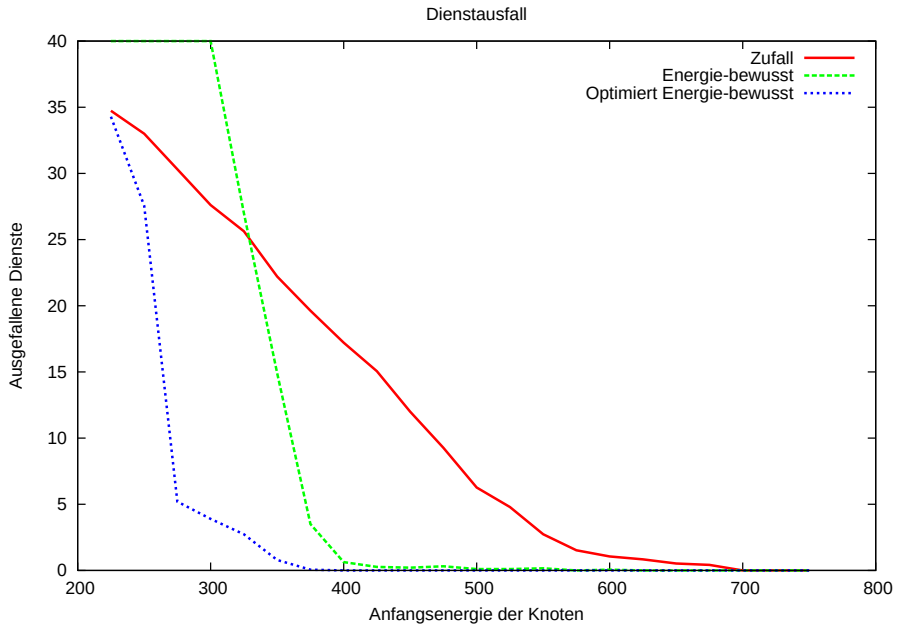

Abbildung 3.34: Ausgefallene Dienste am Ende (für die vollständigen Simulationsparameter siehe auch Tab. A.1 im Anhang)

Bei einer Anfangsenergie von 360 fällt kein Dienst mehr aus, was bedeutet, dass die Anwendung bis zum Ende der Simulationszeit funktionsfähig ist. Im Gegensatz dazu ist die Lebenszeit der Anwendung beim „Energie-bewussten“ Verfahren bis zu einer Anfangsenergie von 420 kürzer als die Simulationszeit. Beim Verfahren „Zufall“ überlebt die Anwendung sogar bis zu einer Anfangsenergie von 700 die Simulationszeit nicht.

Zusammenfassung

In der quantitativen Untersuchung wurde belegt, wie sich die positiven Eigenschaften von PanTalassa bezüglich der freien Verteilbarkeit und der deklarativen Definition der Dienste nutzen lassen, um die Lebenszeit der einzelnen Sensorknoten und insbesondere der verteilten Anwendung erheblich zu steigern. Wenn die Möglichkeit genutzt wird, dass Dienste während der Laufzeit der Anwendung ihre Knotenzuordnung wechseln, kann trotz des damit zusätzlichen Energieaufwands die Zahl der ausgefallenen Dienste erheblich reduziert werden. Wird zusätzlich bereits bei der Initialisierung auf eine optimierte Knotenzuordnung geachtet, sinkt der Gesamtenergieverbrauch signifikant.

3.6.3 Einsatz von PanTalassa in einem realen Sensornetz

Um die Einsatzfähigkeit von TALASSA auch in der Realität zu belegen, wird das Leitszenario „Intelligentes Gewächshaus“ in einem realen Sensornetz als Demonstrator umgesetzt. Im Folgenden werden die verwendete Hardware und Software sowie das zu verwirklichende Szenario vorgestellt.

Im darauffolgenden Abschnitt 3.6.4 wird das Szenario in ein dienstorientiertes Modell überführt. Die Modellierung der verteilten Anwendung „Intelligentes Gewächshaus“ demonstriert die Verwendung der verschiedenen Diensttypen einschließlich virtueller Sensoren und Aktoren, Komposition von Diensten, Änderung und Erweiterung einer Anwendung durch nachträglich ausgebrachte Sensorknoten und Dienste, graphische Darstellung von Diensten, und Visualisierung einer verteilten Anwendung durch Dienstmengen.

Aufbau des Demonstrators „Intelligentes Gewächshaus“ mit verwendeter Hardware und Software

Das Gewächshaus enthält eine Reihe von Pflanzen, die einzeln mit der jeweils benötigten Menge Wasser versorgt werden und mit einer für alle Pflanzen gemeinsam zur Verfügung stehenden Lichtquelle beleuchtet werden. Als Sensoren stehen dabei jeder Pflanze ein Feuchtigkeitssensor zur Verfügung und dem Gewächshaus ein einzelner Lichtsensor. Als Feuchtigkeitsaktor dient eine für jede Pflanze einzeln zur Verfügung stehende Wasserpumpe. Eine Lampe beleuchtet als Helligkeitsaktor die Pflanzen in der Gesamtheit. In Abb. 3.35 ist der Aufbau des Demonstrators „Intelligentes Gewächshaus“ im Überblick zu sehen. Eine einzelne Pflanze mit Sensorknoten mit angeschlossenem Feuchtigkeitssensor und Feuchtigkeitsaktor (Wasserpumpe) ist in Abb. 3.36 abgebildet. Für Demonstrationszwecke wurde auf den Einsatz eines realen Feuchtigkeitssensors verzichtet und stattdessen ein „virtueller“ Feuchtigkeitssensor verwendet. Dazu wurde auf einem PDA (für engl. *Personal Digital Assistant*) eine Anwendung implementiert, die über die serielle Schnittstelle eines Sensorknotens einen Sensorwert auf diesem setzen kann. Im Gegensatz zu einem realen Sensor bietet dieser „virtuelle Feuchtigkeitssensor“ die Möglichkeit, jederzeit beliebige Feuchtigkeitswerte zu melden, um so den Regelkreis besser demonstrieren zu können. Als Aktor für die Bewässerung dient eine mit einem Elektromotor betriebene Wasserpumpe, die vom Sensorknoten an- und ausgeschaltet werden kann. Für die Helligkeitsmessung wird der auf den Sensorknoten befindliche

Abbildung 3.35: Aufbau des Demonstrators „Intelligentes Gewächshaus“

Helligkeitssensor verwendet. Zur Beeinflussung der Helligkeit wird eine Lampe verwendet, die vom Sensorknoten gesteuert werden kann.

Zur Umsetzung des Demonstrators wurde die virtuelle Maschine TalassaVM auf MicaZ-Knoten von Crossbow Inc. [Cro] mit dem Sensorbetriebsystem TinyOS [Tin] implementiert. Die drahtlose Kommunikation erfolgt über das Kommunikationsprotokoll ZigBee [Zig11], das in TinyOS implementiert ist. Zum Auffinden der Dienste im Sensornetz wurde das sichere Dienstverzeichnis SCAN (Secure Content Adressable Networks, siehe [Hof08]) verwendet. Bei der Installation der Dienste wird über SCAN die ZigBee-Knotenadresse zusammen mit der *serviceID* des entsprechenden Dienstes abgelegt. Anhand der *serviceID* kann somit über SCAN die Adresse des Sensorknotens ermittelt werden, auf dem der Dienst gespeichert ist.

Szenario

Im Rahmen des „Intelligenten Gewächshauses“ sollen die Bewässerung und die Beleuchtung geregelt werden und außerdem das Hinzufügen neuer Pflanzen mit der entsprechenden Anpassung des Sensornetzes realisiert werden.

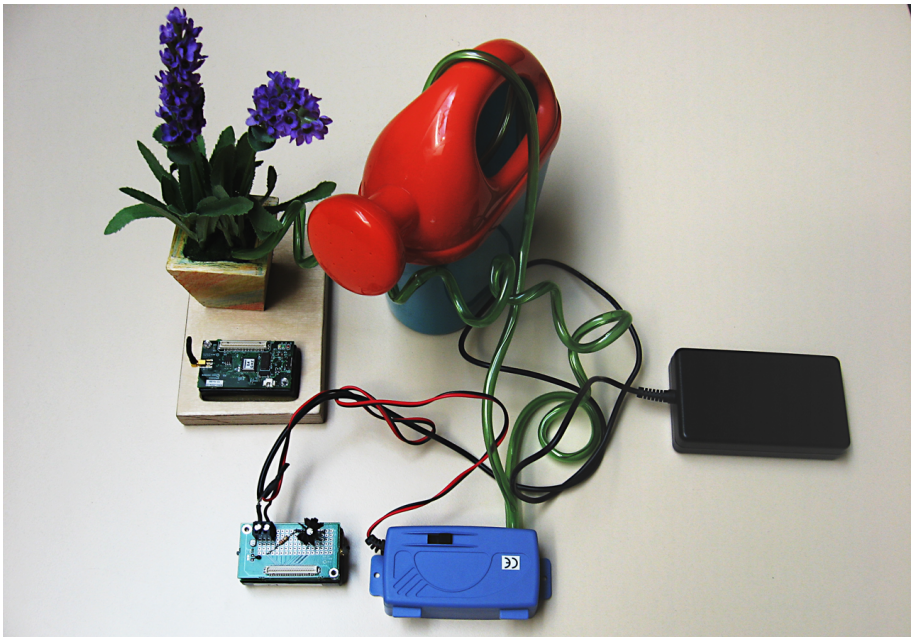

Abbildung 3.36: Einzelne Pflanze mit Sensorknoten und Flüssigkeitsaktor

Feuchtigkeitsregelung

Die Bewässerung soll für jede Pflanze separat geregelt werden. Dazu misst der Feuchtigkeitssensor der Pflanze in regelmäßigen Abständen die Feuchtigkeit. Abhängig von dieser Messung wird im Bedarfsfall die Wasserpumpe eingeschaltet, bis die entsprechende Feuchtigkeit erreicht ist. Wie oben erwähnt, wird keine echte Feuchtigkeitsmessung durchgeführt, sondern die gemessene Feuchtigkeit „manuell“ über ein externes Gerät eingestellt.

Helligkeitsregelung

Die Helligkeitsregelung unterscheidet sich zur Feuchtigkeitsregelung dahingehend, dass das Licht nicht separat für jede Pflanze zur Verfügung steht, sondern alle Pflanzen gemeinsam mit dem gleichen Licht beleuchtet werden. Die unterschiedlichen Bedürfnisse der Pflanzen müssen deswegen gleichzeitig erfüllt werden. Dazu wird jeweils für jede Pflanze eine obere und untere Schranke der erlaubten Helligkeit festgelegt. Die Helligkeitsregelung hält dann die Helligkeit in einem Bereich, der sich aus dem Schnitt aller Intervalle ergibt.

Hinzufügen einer neuen Pflanze

Damit das Gewächshaus erweiterbar ist, sollen neue Pflanzen hinzugefügt werden können. Jede Pflanze bringt auf ihren Sensorknoten die dafür notwendigen Dienste mit. Zum einen sind dies die Dienste zur Feuchtigkeitsregelung, zum anderen sind dies die Dienste mit notwendigen Informationen zur Helligkeitsregelung. Speziell bei der Helligkeitsregelung muss beim Hinzufügen darauf geachtet werden, dass die Werte der neuen Pflanze mit denen der bereits anwesenden Pflanzen vereinbar sind. Vor dem Hinzufügen wird deswegen die Eignung überprüft und eine entsprechende Rückmeldung gegeben. Sind die Werte der neuen Pflanze geeignet, wird sie ins Gewächshaus integriert und die Helligkeitsregelung wird automatisch an die sich neu ergebenden Bedingungen angepasst.

3.6.4 Modellierung der Dienste

Im Folgenden werden die für das oben vorgestellte Szenario notwendigen Dienste modelliert. Nachfolgend werden die primitiven Dienste für die Sensoren und Aktoren entwickelt, die Dienste für die Feuchtigkeitsregelung, die Helligkeitsregelung, die Eignungsprüfung einer Pflanze und das Hinzufügen einer neuen Pflanze.

Sensoren und Aktoren

Die primitiven Dienste für die Feuchtigkeitsregelung und die Helligkeitsregelung teilen sich in *DataRead*-Dienste und *DataWrite*-Dienste auf. Folgende *DataRead*-Dienste werden für die Regelungsaufgaben benötigt:

- Dienst für Feuchtigkeitssensor auf jeder Pflanze,
- Dienste für die Schranken der Feuchtigkeit auf jeder Pflanze,
- Dienst für den Helligkeitssensor im Gewächshaus,
- Dienste für die momentan geltenden Schranken der Helligkeit im Gewächshaus,
- Dienste für die individuellen Schranken der Helligkeit auf jeder Pflanze.

Sämtliche, folgend beschriebenen *DataRead*-Dienste sind in Tab. 3.19 zusammengefasst.

Der Feuchtigkeitssensor wird über den Dienst mit der ID *DR-Humidity-Sensor-Plant(n)* modelliert. Hierbei steht n für die jeweilige Pflanze n. Es gibt somit für

serviceID	Beschreibung	Parameter	Wert
DR-Humidity-Sensor-Plant(n)	Feuchtigkeitssensor der Pflanze n	*driver* *target*	Humidity-Plant(n) initiator
DR-Humidity-Min-Plant(n)	Untere Feuchtigkeitsschranke der Pflanze n	*driver* *target*	Humidity-Min-Plant(n) initiator
DR-Humidity-Max-Plant(n)	Obere Feuchtigkeitsschranke der Pflanze n	*driver* *target*	Humidity-Max-Plant(n) initiator
DR-Light-Sensor-Room	Helligkeitssensor des Raums	*driver* *target*	Light initiator
DR-Light-Min-Room	Untere Helligkeitsschranke des Raums	*driver* *target*	Light-Min-Room initiator
DR-Light-Max-Room	Obere Helligkeitsschranke des Raums	*driver* *target*	Light-Max-Room initiator
DR-Light-Min-Plant(n)	Untere Helligkeitsschranke der Pflanze n	*driver* *target*	Light-Min-Plant(n) initiator
DR-Light-Max-Plant(n)	Obere Helligkeitsschranke der Pflanze n	*driver* *target*	Light-Max-Plant(n) initiator

Tabelle 3.19: *DataRead*-Dienste für Sensoren und gespeicherte Werte im Gewächshausszenario

jede Pflanze, die im Gewächshaus steht, einen solchen Dienst. Dieser soll über einen speziellen Sensortreiber den speziellen Feuchtigkeitssensor dieser Pflanze auslesen. Um kenntlich zu machen, welcher Treiber hierfür genutzt werden soll, hat der Parameter *driver* den Wert *Humidity-Plant(n)*. Die virtuelle Maschine TalassaVM nutzt diese Kennung, um beim Aufruf dieses Dienstes den so gekennzeichneten Sensor auszulesen. Der ausgelesene Wert soll dann dem aufrufenden Dienst zur Verfügung gestellt werden. Somit ist der Parameter *target* auf *initiator* festgelegt.

Für die festgelegten Feuchtigkeitsschranken besitzt jede Pflanze zwei *DataRead*-Dienste, die die Variablen vom Speicher des Sensorknotens auslesen: *DR-Humidity-Min-Plant(n)* und *DR-Humidity-Max-Plant(n)*. Der Parameter *driver* hat hier den Wert *Humidity-Min-Plant(n)* beziehungsweise *Humidity-Max-Plant(n)*. Diese *driver* verweisen in diesem Fall auf einen Treiber, der die im Speicher unter diesem Namen abgelegten Werte zurückgibt. Für die untere Schranke gibt der Treiber dementsprechend den Wert zurück, der dem Speicherplatz *Humidity-Min-Plant(n)* zugeordnet ist, für die obere entsprechend *Humidity-Max-Plant(n)*.

Vergleichbare Dienste gibt es auch für die Helligkeitsregelung: *DR-Light-Sensor-Room*, *DR-Light-Min-Room* und *DR-Light-Max-Room*. Auch hier spricht der erste der aufgezählten Dienste den physikalischen Helligkeitssensor an (*driver* = *Light*) und die letzten beiden geben die entsprechenden Schranken für die Helligkeit zurück. Für die Schranken wird analog zur Feuchtigkeit der Wert im Speicher abgelegt und über die Kennzeichnung *driver* = *Humidity-Min-Plant(n)* und *driver* = *Humidity-Max-Plant(n)* angesprochen.

Zusätzlich zu den Helligkeitsschranken, die für den gesamten Raum gelten, existieren auch individuelle Schranken für die jeweiligen Pflanzen: *DR-Light-Min-Plant(n)* und *DR-Light-Max-Plant(n)*. Diese Dienste befinden sich wie die Pflanzen-individuellen Dienste für die Feuchtigkeitsregelung auf dem Sensorknoten der Pflanze selbst. Die individuellen Schranken werden unter den Speicherplätzen *driver* = *Light-Min-Plant(n)* und *driver* = *Light-Max-Plant(n)* abgelegt und von den oben angegebenen *DataRead*-Diensten aus diesen gelesen.

Für die Aktoren im Gewächshaus-Sensornetz werden folgende *DataWrite*-Dienste benötigt:

- Dienst für die Wasserpumpe jeder Pflanze,
- Dienst für die Pflanzenlampe im Gewächshaus,
- Dienst für die Fehleranzeige auf jeder Pflanze.

serviceID	Beschreibung	Parameter	Wert
DW-Humidity-Actuator-Plant(n)	Wasserpumpe der Pflanze n	*driver*	Pump-Plant(n)
		value	0/1
DW-Light-Actuator-Room	Pflanzenlampe des Raums	*driver*	Lamp
		value	0/1
DW-Error-Plant(n)	LED-Anzeige der Pflanze n	*driver*	LED-red
		value	0/1

Tabelle 3.20: *DataWrite*-Dienste für Aktoren im Gewächshausszenario

Sämtliche, folgend beschriebenen *DataWrite*-Dienste sind in Tab. 3.20 zusammengefasst.

Die Wasserpumpe wird über den Dienst mit der ID *DW-Humidity-Actuator-Plant(n)* angesprochen. Ebenso wie bei den *DataRead*-Diensten steht hier n für die jeweilige Pflanze. Als Parameter *value* kann diesem Dienst 0 oder 1 übergeben werden. Bei *value* = 1 wird die Wasserpumpe angeschaltet, bei *value* = 0 entsprechend gestoppt.

Ähnlich verhält sich der Dienst für die Lampe: *DW-Light-Actuator-Room*. Da im Gewächshaus nur eine einzige Lampe vorhanden ist, existiert auch dieser Dienst genau einmal. Prinzipiell ist die Lampe dafür verantwortlich, fehlende Helligkeit graduell auszugleichen. Geringfügige Änderungen der Helligkeit waren jedoch bei der Demonstration kaum wahrnehmbar. Deswegen wurde auf eine graduelle Helligkeitssteuerung verzichtet und stattdessen ein einfaches An- und Ausschalten realisiert. Wie bei der Wasserpumpe gilt auch bei diesem Aktor für den Parameter *value* die 1 für Einschalten und 0 für Ausschalten der Lampe.

Vor dem tatsächlichen Hinzufügen einer Pflanze wird überprüft, ob die oberen und unteren Helligkeitsschranken mit den Werten der anderen Pflanzen vereinbar sind. Im Fehlerfall kann dies die entsprechende Pflanze über eine rote LED kenntlich machen. Auch für sie gelten für Ein- und Ausschalten die Parameterwert *value* = 1 beziehungsweise *value* = 0.

Dienste der Feuchtigkeitsregelung

Die Feuchtigkeitsregelung soll jede Pflanze mit der für sie geeigneten Menge Wasser versorgen, so dass die Feuchtigkeit des Bodens immer im idealen Bereich bleibt. Dazu

serviceID	Beschreibung	Parameter	Wert
RE-Humidity-Plant(n)	Feuchtigkeits-regelung der Pflanze n	*duration* *interval* *execution list*	-1 30 s CO-HumidityTooLow-Plant(n) CO-HumidityTooHigh-Plant(n)
CO-HumidityTooLow-Plant(n)	Feuchtigkeit zu niedrig?	*operand a* *operator* *operand b* *then-list*	DR-Humidity-Sensor-Plant-(n) $<$ DR-Humidity-Min-Plant(n) DW-Humidity-Actuator-Plant(n) [value=1]
CO-HumidityTooHigh-Plant(n)	Feuchtigkeit zu hoch?	*operand a* *operator* *operand b* *then-list*	DR-Humidity-Sensor-Plant-(n) $>$ DR-Humidity-Max-Plant(n) DW-Humidity-Actuator-Plant(n) [value=0]

Tabelle 3.21: Dienste für Feuchtigkeitsregelung einer Pflanze n

besitzt jede Pflanze einen eigenen Feuchtigkeitssensor und eine eigene Wasserpumpe. Im vorangegangen Abschnitt wurden die für die Nutzdaten zuständigen primitiven Dienste für den Feuchtigkeitssensor, die Wasserpumpe und die jeweils einzuhaltenden Schranken vorgestellt.

Den Kontrollfluss der Feuchtigkeitsregelung modellieren drei komponierende Dienste. Ein *Repetitive*-Dienst führt regelmäßig zwei *Conditional*-Dienste aus, die jeweils die obere und untere Schranke kontrollieren und gegebenenfalls über *DataWrite*-Dienste die Wasserpumpe an- oder abschalten. Die vollständige Beschreibung der Dienste samt deren Parameter findet sich in Tab. 3.21. Die graphische Repräsentation des Dienstmodells ist in Abb. 3.37 abgebildet.

Gestartet wird die Feuchtigkeitsregelung über den Dienst *RE-Humidity-Plant(n)*. Dieser startet regelmäßig in bestimmten Zeitabständen die beiden *Conditional*-Dienste

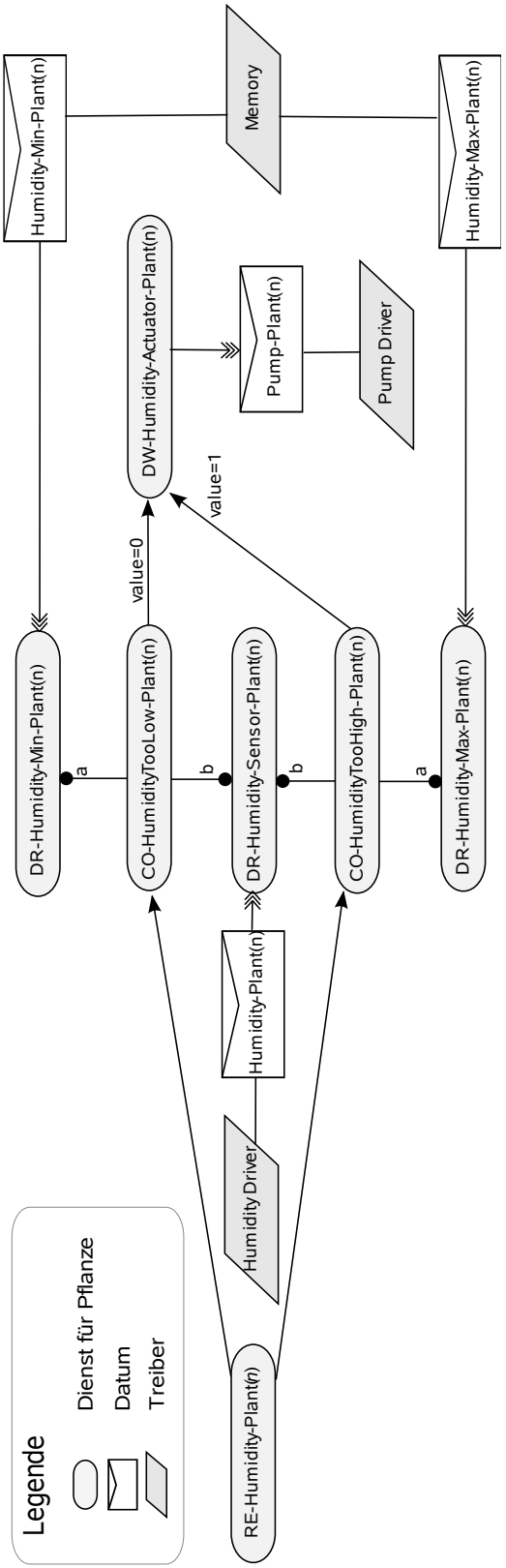

Abbildung 3.37: Feuchtigkeitsregelung

serviceID	Beschreibung	Parameter	Wert
RE-Light-Room	Helligkeits-regelung des Raums	*duration* *interval* *execution list*	-1 30 s CO-LightTooLow-Room CO-LightTooHigh-Room
CO-LightTooLow-Room	Helligkeit zu niedrig?	*operand a* *operator* *operand b* *then-list*	DR-Light-Sensor-Room $<$ DR-Light-Min-Room DW-Light-Actuator-Room [value=1]
CO-LightTooHigh-Room	Helligkeit zu hoch?	*operand a* *operator* *operand b* *then-list*	DR-Light-Sensor-Room $>$ DR-Light-Max-Room DW-Light-Actuator-Room [value=0]

Tabelle 3.22: Dienste für Helligkeitsregelung des Raums

CO-HumidityTooLow-Plant(n) und *CO-HumidityTooHigh-Plant(n)*. Wie bei den primitiven Diensten steht der Platzhalter n für die Nummer der jeweiligen Pflanze n, da jede Pflanze eine individuelle Feuchtigkeitsregelung hat. Die beiden *Conditional*-Dienste vergleichen den aktuellen Wert des Feuchtigkeitssensors (*DR-Humidity-Sensor-Plant(n)*) mit den Soll-Werten (*DR-Humidity-Min-Plant(n)* und *DR-Humidity-Min-Plant(n)*) und schalten bei Bedarf die Wasserpumpen an beziehungsweise aus (*DW-Humidity-Actuator-Plant(n)*). Der Feuchtigkeitssensor wird hierbei über den entsprechenden *driver Humidity-Plant(n)* ausgelesen. Die Soll-Werte sind als virtuelle Sensoren im Speicher abgelegt und werden über den *driver Humidity-Min-Plant(n)* beziehungsweise *Humidity-Max-Plant(n)* ausgelesen.

Dienste der Helligkeitsregelung

Die Helligkeitsregelung ist vom prinzipiellen Aufbau der Feuchtigkeitsregelung sehr ähnlich. Auch hier werden ein Sensor und ein Aktor genutzt, um in einem aus komponierenden Diensten zusammengesetzten Regelkreis einen physikalischen Wert in vorgegebenen Schranken zu halten. Im Unterschied zur Feuchtigkeit gibt es jedoch nicht für jede einzelne Pflanze einen separaten Regelkreis. Die Helligkeit wird für das gesamte Gewächshaus

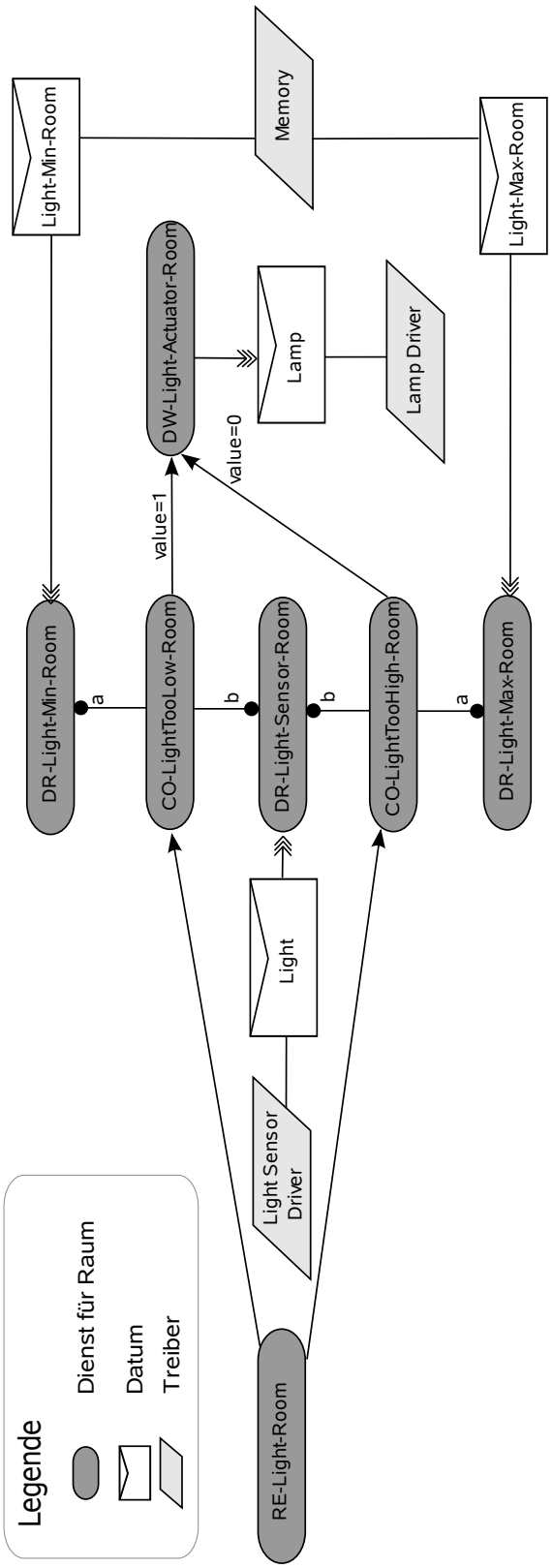

Abbildung 3.38: Helligkeitsregelung

gemessen und eingestellt. Dazu ist im Gewächshaus ein Helligkeitssensor vorhanden. Eine Pflanzenlampe dient als Helligkeitsaktor. Die für die Nutzdaten zuständigen primitiven Dienste für den Helligkeitssensor, die Pflanzenlampe und die geltenden Schranken wurden bereits oben vorgestellt.

Der Regelkreis für die Helligkeit wird über drei komponierende Dienste realisiert. Ein *Repetitive*-Dienst führt regelmäßig zwei *Conditional*-Dienste aus, die jeweils die obere und untere Schranke kontrollieren und bei Bedarf die Intensität der Pflanzenlampe korrigieren. Die vollständige Beschreibung der Dienste samt deren Parameter findet sich in Tab. 3.22. Die graphische Repräsentation des Dienstmodells ist in Abb. 3.38 abgebildet.

Gestartet wird die Helligkeitsregelung über den Dienst *RE-Light-Room*. Dieser startet in regelmäßigen Abständen die beiden *Conditional*-Dienste *CO-LightTooLow-Room* und *CO-LightTooHigh-Room*. Stellt der *Conditional*-Dienst *CO-LightTooLow-Room* eine zu geringe Helligkeit fest, wird das Licht der Pflanzenlampe über den *DataWrite*-Dienst *DW-Light-Actuator-Room* eingeschaltet. Dementsprechend wird das Licht bei Bedarf durch den *Conditional*-Dienst *CO-LightTooHigh-Room* ausgeschaltet. Der Helligkeitssensor wird mit dem *DataRead*-Dienst *DR-Light-Sensor-Room* über den zugehörigen *driver Light* ausgelesen. Die Soll-Werte für die Helligkeit sind im Speicher abgelegt und werden mit den *DataRead*-Diensten *DR-Light-Min-Room* und *DR-Light-Max-Room* über den *driver Light-Min-Room* und *Light-Max-Room* ausgelesen.

Dienste für die Eignungsprüfung einer neuen Pflanze

Das Gewächshaus soll mit Pflanzen erweitert werden können. Da jede Pflanze ihre individuelle Feuchtigkeitsregelung mit den entsprechenden Diensten mitbringt, stellt dies keine Herausforderung für das Sensornetz dar. Anders verhält es sich bei der Helligkeitsregelung. Diese basiert auf Schranken, die für alle Pflanzen im gleichen Maße gelten, da das Gewächshaus nur eine Pflanzenlampe besitzt. Somit ist auch die Helligkeit für alle Pflanzen gleich. Bevor eine neue Pflanze ins Gewächshaus gebracht wird, muss folglich überprüft werden, ob die neue Pflanze mit ihren individuellen Lichtbedürfnissen mit dem bereits durch Schranken festgelegten Lichtverhältnis für die anderen Pflanzen harmonisiert. Die Pflanze eignet sich, wenn ihre individuellen Schranken eine nichtleere Schnittmenge mit den bereits vorhandenen Schranken im Gewächshaus besitzen. Ungeeignet ist die Pflanze demnach, wenn ihre obere Schranke unter der vorhandenen unteren Schranke liegt, beziehungsweise wenn ihre untere Schranke oberhalb der oberen vorhandenen Schranke liegt.

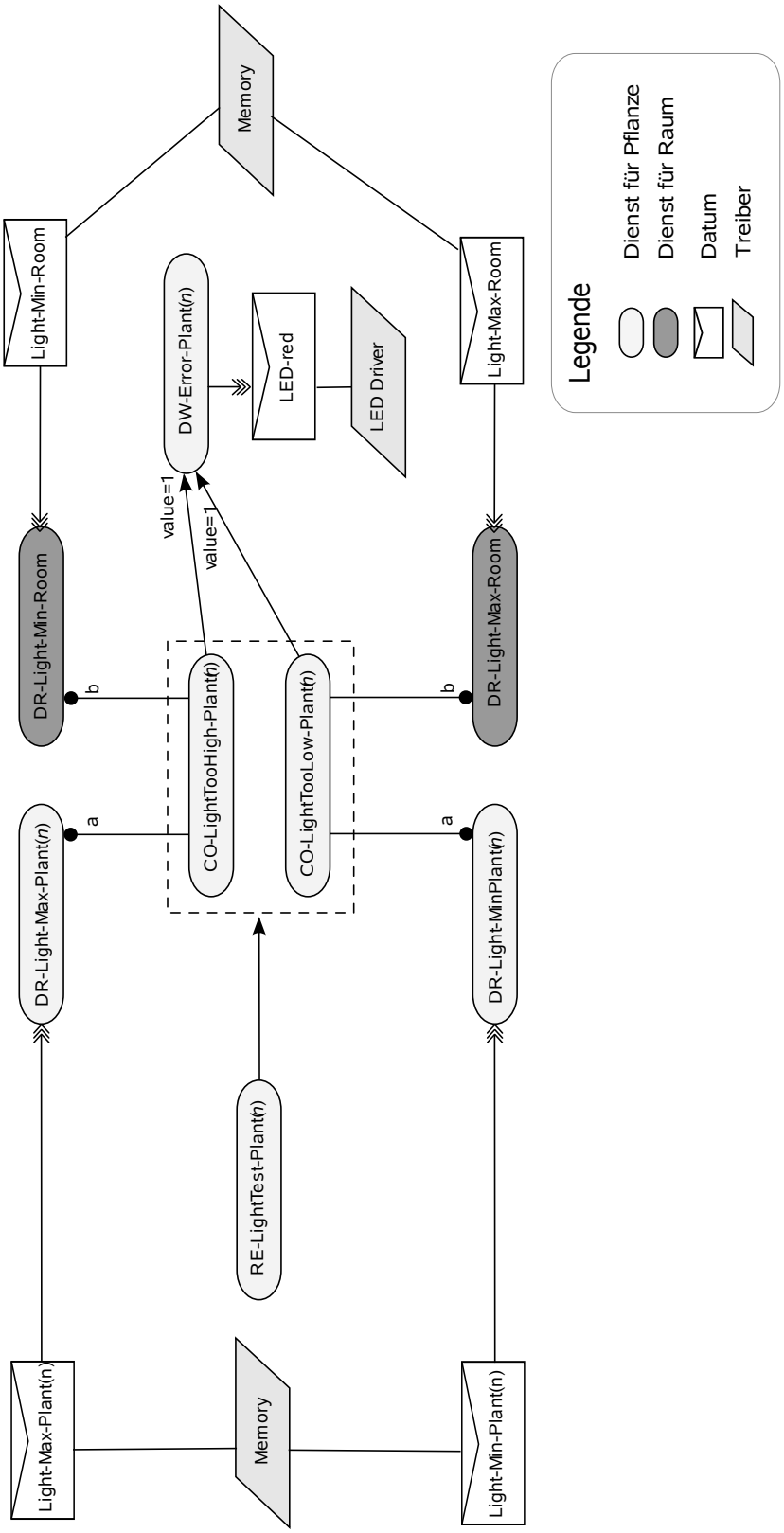

Abbildung 3.39: Prüfung auf Eignung der neuen Pflanze

serviceID	Beschreibung	Parameter	Wert
RE-LightTest-Plant(n)	Prüfung auf die Eignung der Pflanze	*duration* *interval* *execution list*	1 s 1 s CO-LightTooLow-Plant(n) CO-LightTooHigh-Plant(n)
CO-LightTooLow-Plant(n)	Helligkeits-schranke des Raums zu klein?	*operand a* *operator* *operand b* *then-list*	DR-Light-Min-Plant(n) > DR-Light-Max- Room DW-Error-Plant(n) [value=1]
CO-LightTooHigh-Plant(n)	Helligkeits-schranke des Raums zu groß?	*operand a* *operator* *operand b* *then-list*	DR-Light-Max-Plant(n) < DR-Light-Min-Room DW-Error-Plant(n) [value=1]

Tabelle 3.23: Dienste für Prüfung auf Eignung der Pflanze für den Raum

Jede Pflanze hat für die Eignungsprüfung ein individuelles Set von Diensten. Als primitive Dienste werden die gewünschten Helligkeitsschranken der Pflanze als *DataRead*-Dienste genutzt. Für eine etwaige Fehlermeldung dient ein *DataWrite*-Dienst zum Anschalten einer roten LED. Zusätzlich werden die *DataRead*-Dienste der aktuell geltenden Helligkeitsschranken im Gewächshaus genutzt. Drei komponierende Dienste implementieren den Kontrollfluss der Eignungsprüfung. Die vollständige Beschreibung der Dienste samt deren Parameter findet sich in Tab. 3.23. Die graphische Repräsentation des Dienstmodells ist in Abb. 3.39 abgebildet.

Gestartet wird die Eignungsprüfung mit dem *Repetitive*-Dienst *RE-LightTest-Plant(n)*. Dieser startet einmalig zwei *Conditional*-Dienste, die jeweils die obere, beziehungsweise untere Schranke auf Eignung überprüfen. Zur Prüfung der oberen Schranke vergleicht der *Conditional*-Dienst *CO-LightTooHigh-Plant(n)* die obere Schranke der Pflanze (*DR-Light-Max-Plant(n)*) mit der unteren Schranke des Gewächshauses (*DR-Light-Min-Room*). Sollte diese kleiner sein als jene, wird über den *DataWrite*-Dienst *DW-Error-Plant(n)* eine rote LED auf der Pflanze angeschaltet. Entsprechend verläuft die Eignungsprüfung der unteren Schranke der Pflanze. Wird in einem der beiden Fälle festgestellt, dass die Pflanze ungeeignet ist, leuchtet die rote LED auf und die Pflanze

serviceID	Beschreibung	Parameter	Wert
DW-New-Light-Min	Neue untere Schranke für die Helligkeit	*driver* *value*	Temp-Min DR-Light-Min-Plant(n)
DR-New-Light-Min	Lesen der neuen unteren Schranke	*driver* *target*	Temp-Min initiator
DW-New-Light-Max	Neue obere Schranke für die Helligkeit	*driver* *value*	Temp-Max DR-Light-Max-Plant(n)
DR-New-Light-Max	Lesen der neuen oberen Schranke	*driver* *target*	Temp-Max initiator

Tabelle 3.24: Dienste für Schreiben und Lesen der neuen Schranken für die Helligkeit

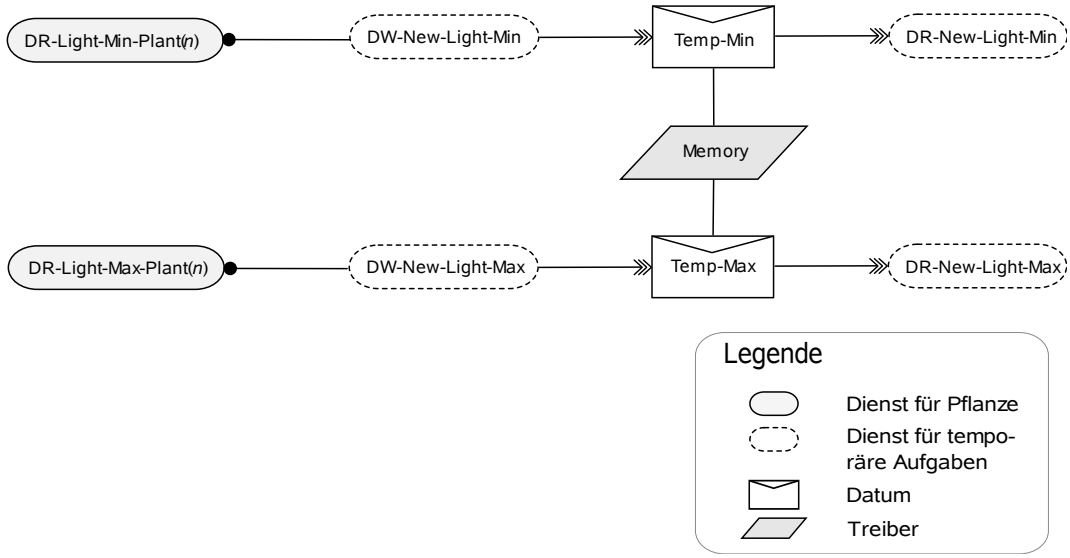

Abbildung 3.40: Temporäre Dienste für Schranken der neuen Pflanze

kann dem Gewächshaus nicht hinzugefügt werden. Andernfalls kann mit dem Hinzufügen begonnen werden. Dies wird im folgenden Abschnitt erläutert.

Dienste für das Hinzufügen einer neuen Pflanze

Die Dienste für das Hinzufügen sollen die Integration einer neuen Pflanze ins Gewächshaus übernehmen. Es wird davon ausgegangen, dass die zuvor beschriebene Eignungsprüfung erfolgreich verlaufen ist. Beim Hinzufügen sollen dann neue Schranken für die

Helligkeitsregelung gefunden werden, und die Helligkeitsregelung des Gewächshauses entsprechend angepasst werden.

Das Hinzufügen einer Pflanze stellt das Sensornetz vor eine besondere Herausforderung. Die neu gewünschten Schranken der Pflanze müssen vom bereits etablierten Gewächshaus-Sensornetz erkannt werden. Da die Parameter der Dienste im Gewächshaus aber bereits festgelegt sind, müssen die neuen Schranken über einen schon bekannten Dienstnamen gelesen werden. Zur Lösung dieses Problems wurden die neuen Schranken als temporäre Dienste realisiert. Vor der Ermittlung der neuen Gewächshausschranken werden somit die neuen Schranken der Pflanze über namentlich bereits bekannte Dienste zur Verfügung gestellt. Die vollständige Beschreibung der Dienste samt deren Parameter findet sich in Tab. 3.24. Es gibt vier temporäre Dienste, jeweils für jede Schranke einen *DataWrite*-Dienst zum Schreiben der neuen Schranken und einen *DataRead*-Dienst zum Auslesen dieser Schranken. Die graphische Repräsentation des Dienstmodells ist in Abb. 3.40 abgebildet.

Vor dem Hinzufügen werden die temporären Dienste (z. B. der Pflanze m) gelöscht und die namentlich gleichen, aber mit anderen Parametern belegten, temporären Dienste (z. B. der Pflanze n) ins Sensornetz eingebracht. Mit dem Aufrufen der beiden *DataWrite*-Dienste (*DW-New-Light-Min* und *DW-New-Light-Max*) werden die neuen Schranken der Pflanze n ausgelesen und unter *Temp-Min* beziehungsweise *Temp-Max* abgespeichert. Das Gewächshaus kann anschließend über die beiden *DataRead*-Dienste (*DR-New-Light-Min* und *DR-New-Light-Max*) die Schranken wieder auslesen.

Nachdem die neuen Schranken der Pflanze dem Sensornetz unter bekannten Dienstnamen zur Verfügung stehen, kann die neue Pflanze schließlich hinzugefügt werden. Dabei werden die gewünschten Helligkeitsschranken der Pflanze mit den bestehenden Helligkeitsschranken des Gewächshauses verglichen und ggf. angepasst. Wenn die untere gewünschte Helligkeitsschranke der Pflanze höher ist als die bestehende des Gewächshauses, muss diese bestehende auf den Wert der neuen Pflanze angehoben werden. Sollte dies nicht der Fall sein, bleibt die bestehende Schranke auf ihrem alten Wert. Mit entsprechender Anpassung läuft diese Prüfung und die eventuell notwendige Korrektur auch für die obere gewünschte Schranke. Der Vergleich der Schranken wird über *Conditional*-Dienste durchgeführt, die die passenden *DataRead*-Dienste als Parameter haben. Die Schranken werden über *DataWrite*-Dienste geändert. Die vollständige Beschreibung der Dienste samt deren Parameter findet sich in Tab. 3.25. Die graphische Repräsentation des Dienstmodells ist in Abb. 3.41 abgebildet.

serviceID	Beschreibung	Parameter	Wert
RE-LightAdjustment-Room	Anpassung der Schranken für Licht	*duration* *interval* *execution list*	1 1 s CO-Light-Min-Adjustment-Room CO-Light-Max-Adjustment-Room
CO-Light-Min-Adjustment-Room	Untere Helligkeitsschranke anpassen	*operand a* *operator* *operand b* *then-list*	DR-Light-Min-Room $<$ DR-New-Light-Min DW-Light-Min-Room
CO-Light-Max-Adjustment-Room	Obere Helligkeitsschranke anpassen	*operand a* *operator* *operand b* *then-list*	DR-Light-Max-Room $>$ DR-New-Light-Max DW-Light-Max-Room
DW-Light-Min	Neue untere Schranke für Helligkeit	*driver* *value*	Light-Min-Room DR-New-Light-Min
DW-Light-Max	Neue obere Schranke für Helligkeit	*driver* *value*	Light-Max-Room DR-New-Light-Max

Tabelle 3.25: Dienste für die Schrankenanpassung der Lichtregelung

Abbildung 3.41: Anpassung der neuen Schranken für die Lichtregelung

serviceID	Beschreibung	Parameter	Wert
RE-PlantAddition-Room	Hinzufügen einer neuen Pflanze	*duration* *interval* *execution list*	1 1 s DW-New-Light-Min DW-New-Light-Max RE-Light-Adjustment-Room

Tabelle 3.26: Dienste für Hinzufügen einer neuen Pflanze ins Gewächshaus

Gestartet wird das Hinzufügen über den *Repetitive*-Dienst *RE-LightAdjustment-Room*, der zwei *Conditional*-Dienste startet. Der erste *Conditional*-Dienst *CO-Light-Min-Adjustment-Room* hat die Operanden *DR-Light-Min-Room*, der die aktuell geltende untere Helligkeitsschranke des Gewächshauses bereitstellt, und *DR-New-Light-Min*, der die gewünschte untere Schranke der gerade hinzuzufügenden Pflanze enthält. Ergibt der Vergleich, dass die gewünschte untere Schranke höher ist als die bestehende, wird der *DataWrite*-Dienst *DW-Light-Min-Room* gestartet. Dieser *DataWrite*-Dienst hat den gewünschten Schrankenwert als Parameter und schreibt diesen Wert im Speicher unter dem Namen *Light-Min-Room*. Die Helligkeitsregelung greift auf diesen Wert zu, um die Helligkeit in den gegebenen Schranken zu halten. Die Anpassung der oberen Schranke des Gewächshauses verläuft analog mit den entsprechend angepassten Diensten, Parametern und Werten.

Die Dienste für die Anpassung der Schranken bilden folglich beim Hinzufügen jeder Pflanze den Schnitt aus den gewünschten Pflanzenschranken und den bestehenden Gewächshausschranken.

Das Hinzufügen der Pflanze besteht aus den Schritten „Schreiben der neuen gewünschten Helligkeitsschranken“ und ggf. der „Anpassung der Helligkeitsschranken des Gewächshauses“. Diese Schritte werden durch einen *Repetitive*-Dienst durchgeführt. Die vollständige Beschreibung dieses Dienstes samt seiner Parameter findet sich in Tab. 3.26. Die graphische Repräsentation des Dienstmodells ist in Abb. 3.42 abgebildet.

Im Folgenden werden alle vorgestellten Dienstmodelle zu einem einzigen Modell zusammengefasst und gesamtheitlich beschrieben.

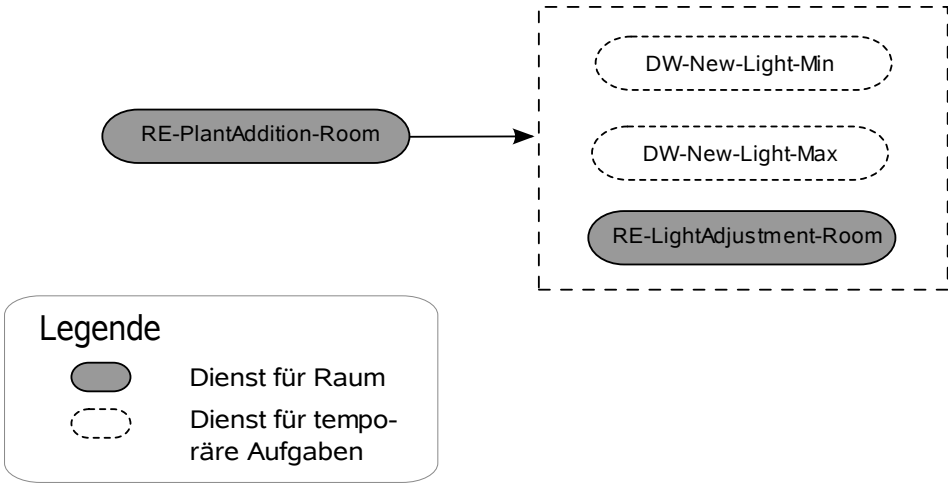

Abbildung 3.42: Hinzufügen einer neuen Pflanze ins Gewächshaus

Komplettes Szenario

In den vorangegangen Abschnitten wurde beschrieben, welche primitiven Dienste zur Verfügung stehen, wie die Feuchtigkeits- und Helligkeitsregelung modelliert sind und wie das Hinzufügen einer Pflanze modelliert ist. Um einen gesamtheitlichen Blick zu ermöglichen, werden diese Dienstmodelle zu Dienstmengen zusammengefasst. Bei allen folgend beschriebenen Dienstmengen werden die möglichen Eingänge definiert. Dies umfasst sowohl die Dienste, die von anderen Dienstmengen gestartet werden können, als auch die Parameter und Daten, die von anderen Dienstmengen genutzt werden. Die Ausgänge umfassen sowohl die Dienste, die bei anderen Dienstmengen gestartet werden, wie auch die Daten und Parameter, die anderen Diensten zur Verfügung gestellt werden.

In Abb. 3.43 sind die Dienstmengen für die Feuchtigkeitsregelung, die Helligkeitsregelung und die Eignungsprüfung einer neuen Pflanze dargestellt.

Für die Feuchtigkeitsregelung sind die Dienste zusammengefasst, die die Wasserzufuhr für jede Pflanze regeln. Sie bilden die Dienstmenge „Humidity Regulation“. Da jede Pflanze ihre eigene Feuchtigkeitsregelung besitzt, existiert auch für jede Pflanze im Gewächshaus eine eigene solche Dienstmenge. Die Feuchtigkeitsregelung kann über den Dienst *RE-Humidity-Plant(n)* gestartet werden, wobei n für die jeweilige Pflanze n steht.

Die Lichtregelung existiert im gesamten Gewächshaus nur einmal. Alle für sie notwendigen Dienste werden in der Dienstmenge „Light Regulation“ zusammengefasst. Sie wird

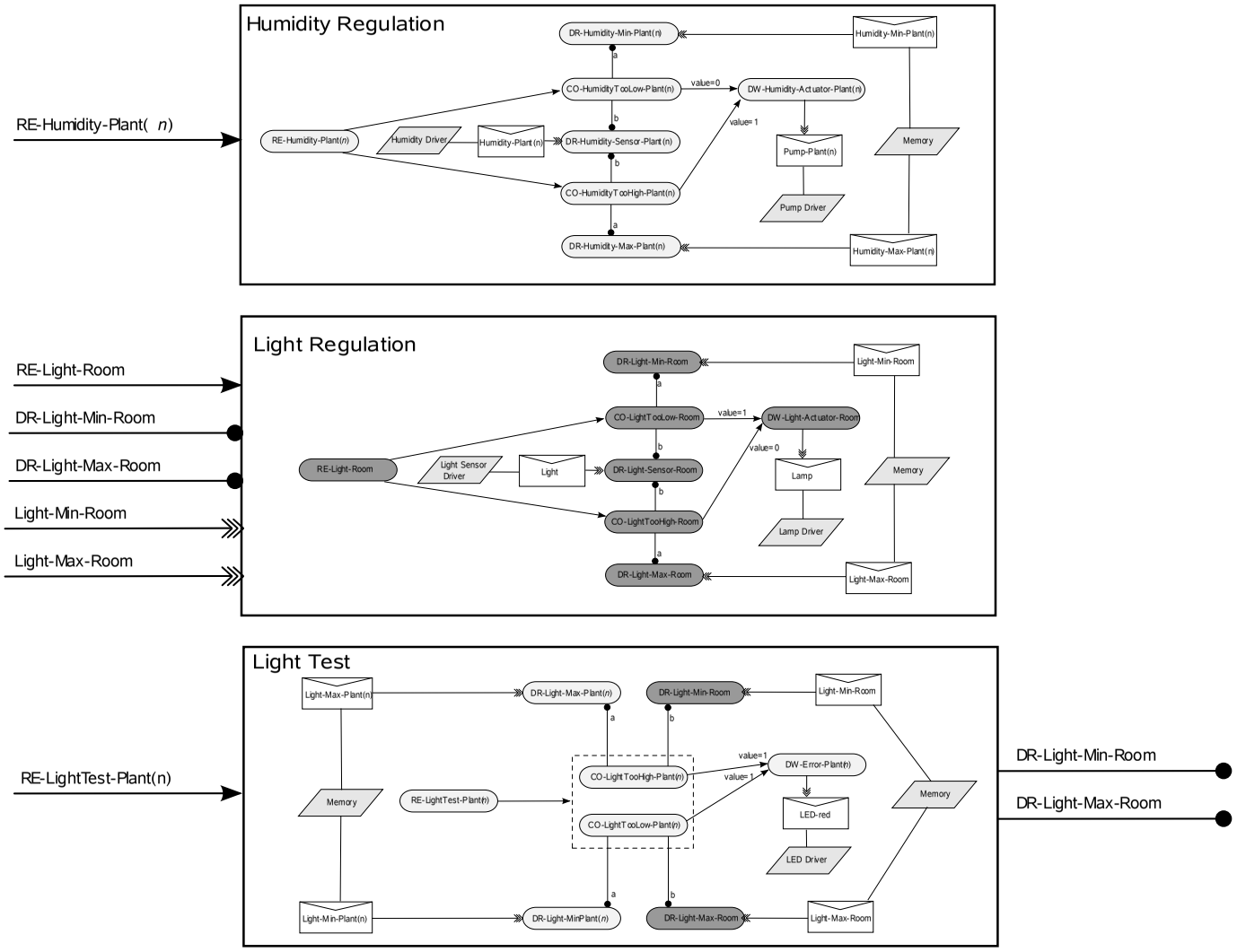

Abbildung 3.43: Modellierung der Feuchtigkeitsregelung, Helligkeitsregelung und Eignungsprüfung als Dienstmengen

über den Dienst *RE-Light-Room* gestartet. Als zusätzliche Eingangsparameter besitzt die Lichtregelung die Daten *Light-Min-Room* und *Light-Max-Room*, die die geltenden Helligkeitsschranken enthalten. Über diese Daten können die Helligkeitsschranken von anderen Dienstmengen geändert werden. Diese Helligkeitsschranken werden von den *DataRead*-Diensten *DR-Light-Min-Room* und *DR-Light-Max-Room* ausgelesen. Diese Dienste dienen anderen Dienstmengen als Parameter.

In der Dienstmenge „Light Test“ sind die Dienste für die Eignungsprüfung einer neuen Pflanze zusammengefasst. Sie wird über den *Repetitive*-Dienst *RE-LightTest-Plant(n)* gestartet. Als Parameter werden die *DataRead*-Dienste *DR-Light-Min-Room* und *DR-Light-Max-Room* genutzt, die die aktuellen Werte der Helligkeitsschranken zur Verfügung stellen.

In Abb. 3.44 sind die Dienstmengen für das Anpassen der Helligkeitsschranken, die gewünschten Helligkeitsschranken einer Pflanze und das Hinzufügen einer Pflanze dargestellt.

In der Dienstmenge „Light Threshold“ sind die Dienste zusammengefasst, die für das Anpassen der Helligkeitsschranken im Gewächshaus nötig sind. Gestartet wird die Dienstmenge über den *Repetitive*-Dienst *RE-LightAdjustment-Room*. Als Parameter dienen die *DataRead*-Dienste für die geltenden Helligkeitsschranken des Gewächshauses (*DR-Light-Min-Room* und *DR-Light-Max-Room*) und für die gewünschten Helligkeitsschranken einer neuen Pflanze (*DR-New-Light-Min* und *DR-New-Light-Max*). Bei der Anpassung der Helligkeitsschranken verändert die Dienstmenge die Daten *Light-Min-Room* und *Light-Max-Room*.

Die Dienstmenge „New Light Values“ fasst die Dienste zusammen, die die neuen gewünschten Helligkeitsschranken einer Pflanze enthalten. Gestartet werden können die *DataWrite*-Dienste *DW-New-Light-Min* und *DW-New-Light-Max*, mit denen die neuen Werte der hinzuzufügenden Pflanze gespeichert werden. Diese können über die *DataRead*-Dienste *DR-New-Light-Min* und *DR-New-Light-Max* ausgelesen werden und anderen Dienstmengen als Parameter dienen.

Die Dienstmenge „Plant Addition“ enthält nur einen einzigen Dienst, der die für das Hinzufügen einer neuen Pflanze erforderlichen Dienste startet. Die Dienstmenge wird über den *Repetitive*-Dienst *RE-PlantAddition-Room* gestartet. Sie selbst startet die Dienste *DW-New-Light-Min*, *DW-New-Light-Max* und *RE-LightAdjustment-Room*.

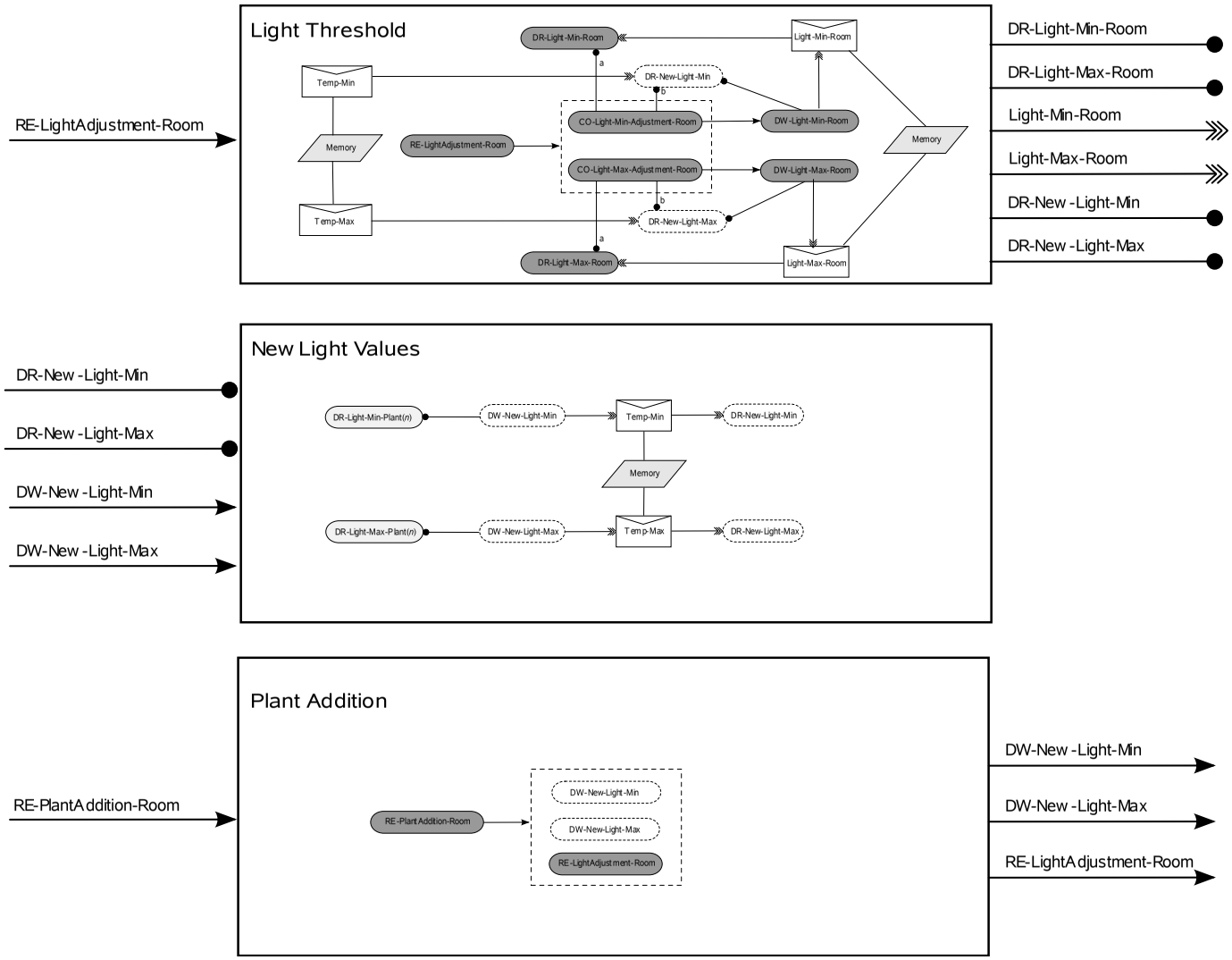

Abbildung 3.44: Modellierung neuer Lichtwerte, Schrankenanpassung und Hinzufügen einer Pflanze als Dienstmengen

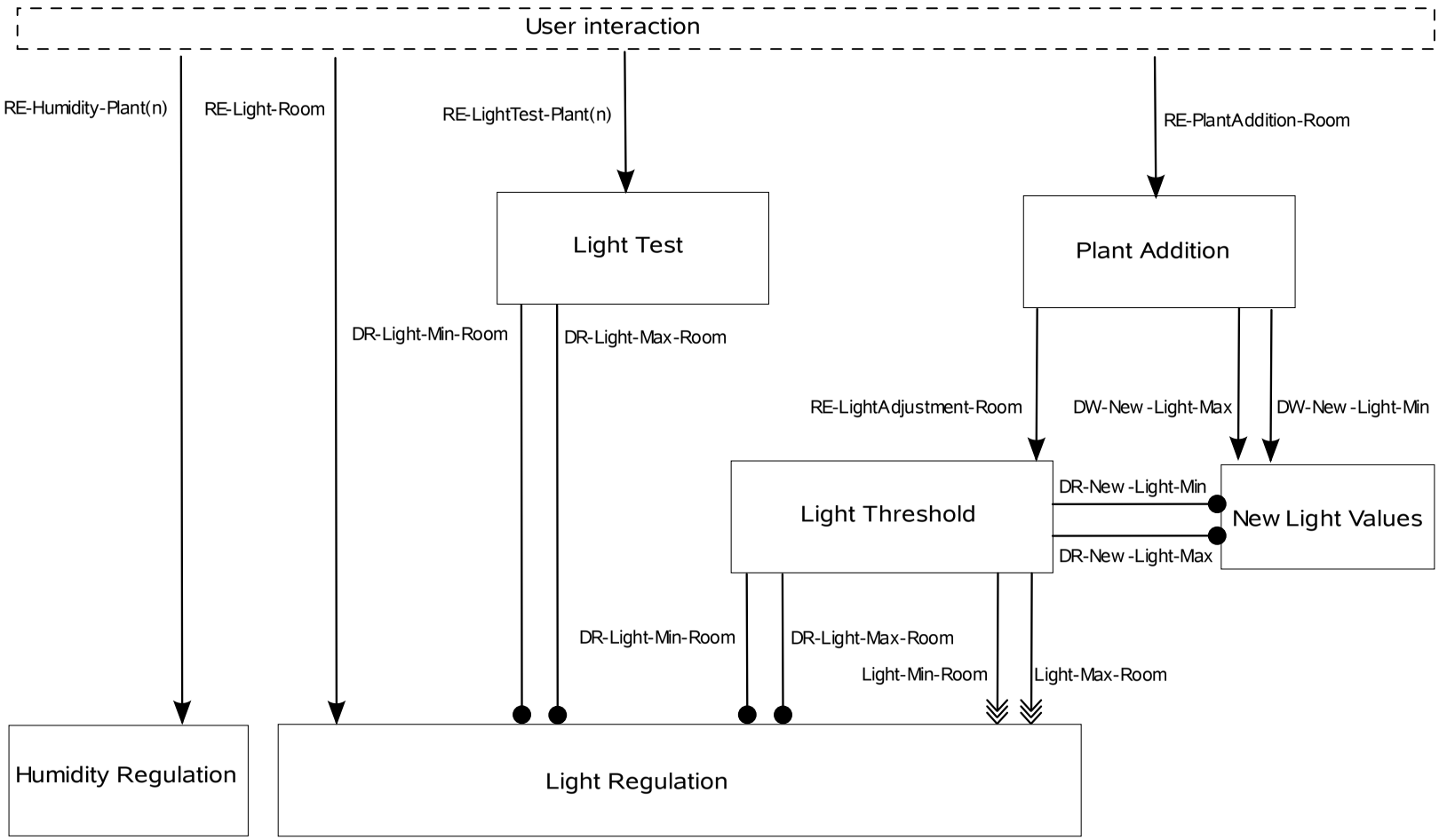

Abbildung 3.45: Komplettes Modell des intelligenten Gewächshauses

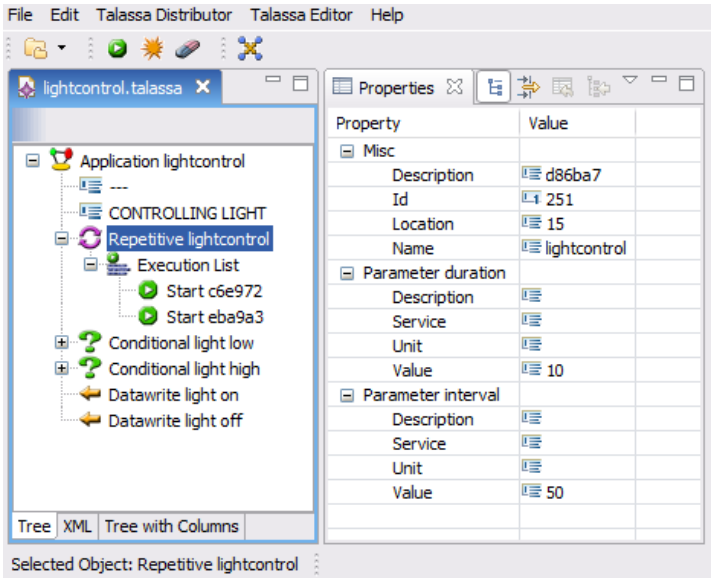

Abbildung 3.46: TalassaDisposer

Die zuvor vorgestellten Dienstmengen werden jetzt in einem Modell vereinigt, das dem kompletten Modell des Gewächshausszenarios entspricht. In Abb. 3.45 ist dieses Modell dargestellt.

Die vorgesehenen Interaktionsmöglichkeiten des Benutzers sind im oberen Bereich zusammengefasst. Über eine geeignete Benutzerschnittstelle kann der Gärtner die Feuchtigkeitsregelung jeder Pflanze (*RE-Humidity-Plant(n)*), die Helligkeitsregelung des Gewächshauses (*RE-Light-Room*), die Eignungsprüfung einer hinzuzufügenden Pflanze (*RE-LightTest-Plant(n)*) und das Hinzufügen einer Pflanze (*RE-PlantAddition-Room*) starten.

Das hier vorgestellte komplette Modell wurde im Rahmen einer Studienarbeit [Her07] auf realen Sensorknoten implementiert. Als Benutzerschnittstelle wurde dort auch eine Anwendung „TalassaDisposer“ entwickelt, mit der sich sowohl Dienstbeschreibungen erstellen als auch Dienste ins Sensornetz ausbringen und Dienste starten lassen. Ein Bildschirmabzug dieser Anwendung ist in Abb. 3.46 zu sehen. Mithilfe dieser Benutzerschnittstelle kann das Szenario des intelligenten Gewächshauses komplett demonstriert werden. Der typische Ablauf einer Demonstration des intelligenten Gewächshauses gliedert sich, wie in der folgenden Aufzählung beschrieben. Dabei wird der Einfachheit halber davon ausgegangen, dass sämtliche Dienste bereits auf dem TalassaDisposer vorliegen und die Helligkeitsregelung (Dienstmenge „Light Regulation“) bereits im

Gewächshaus-Sensornetz installiert und gestartet wurden. Als „Gärtner“ wird die Person bezeichnet, die für die manuellen Interaktionen verantwortlich ist.

1. Eine neue Pflanze mit ihrem individuellen Sensorknoten wird zum Gewächshaus gestellt. Die erforderlichen Dienste auf diesem Sensorknoten installiert der Gärtner mithilfe des TalassaDisposers (Dienste für die Feuchtigkeitsregelung, Helligkeitsschranken der Pflanze).
2. Mithilfe des TalassaDisposers startet der Gärtner die Eignungsprüfung (Dienstmenge „Light Test“). Stellt sich die Pflanze als ungeeignet heraus, da die Helligkeitsschranken nicht mit den bestehenden vereinbar sind, leuchtet auf dem Sensorknoten eine rote LED auf.
3. War die Eignungsprüfung erfolgreich, installiert der Gärtner die temporären Dienste für die Helligkeitsschranken im Sensornetz. Damit sind die neuen gewünschten Schranken dem Sensornetz bekannt.
4. Der Gärtner startet das Hinzufügen einer neuen Pflanze (Dienstmenge „Plant Addition“). Durch diese werden die Schranken ggf. angepasst. Die Helligkeitsregelung basiert dann auf den neuen Werten.
5. Der Gärtner startet die Feuchtigkeitsregelung der Pflanze (Dienstmenge „Humidity Regulation“).
6. Die Pflanze wurde erfolgreich hinzugefügt, und eine weitere Pflanze kann beginnend mit Schritt 1 hinzugefügt werden.

Zusammenfassung

Die Implementierung des Leitszenarios „Intelligentes Gewächshaus“ hat gezeigt, dass Talassa auf realen Sensorknoten einsetzbar und realisierbar ist. Das gesamte Szenario für Licht- und Feuchtigkeitsregelung konnte mit den zur Verfügung stehenden Diensttypen von Talassa modelliert und implementiert werden. Auch die flexible Erweiterung und Änderung einer Anwendung durch nachträglich ausgebrachte Sensorknoten und Dienste konnte im „Intelligent Gewächshaus“ demonstriert werden, indem die Licht- und Feuchtigkeitsregelung für neu hinzukommende Pflanzen angepasst wurden. Neue Pflanzen brachten hierzu neue Dienste mit, die dann im Sensornetz integriert wurden beziehungsweise bestehende Dienste änderten.

Zur Modellierung des Szenarios wurde die für Talassa entwickelte graphische Notation dienstorientierter Anwendungen genutzt. Sie ermöglichten eine übersichtliche

Darstellung aller Teilaspekte des „Intelligenten Gewächshauses“. Ebenso konnte das komplette Szenario durch Zusammenfassung der Teilaspekte zu Dienstmengen ganzheitlich und übersichtlich visualisiert werden.

Das so beschriebene Szenario wurde erfolgreich international demonstriert und vorgestellt [HHH+07]. Der grundlegende Aufbau von Talassa wurde in [HZ04] [HHZ04] international veröffentlicht.

4 Modusbasierte Optimierung der Kommunikation

Sensornetze können für viele verschiedene Anwendungsfelder genutzt werden, z. B. zur Unterstützung und Hilfe eingeschränkter oder kranker Menschen in ihrer Wohnung, im intelligenten Haus oder in intelligenten Büroumgebungen. Drahtlose Sensornetze dienen dabei der Beobachtung von Körperfunktionen und der Umgebungssituation, können physikalische Größen wie Licht oder Temperatur regeln, oder sie können auch als Alarmsysteme genutzt werden, um gefährliche Zustände bei Menschen oder in Räumen zu melden. In Abhängigkeit der Aufgabe und der Umgebungssituation ist das Sensornetz somit zuständig, Daten energieeffizient zu sammeln oder auch im Alarmfall Daten möglichst schnell an entsprechende Datensenken weiterzuleiten.

In diesem Kapitel wird ein Modus-Rahmenwerk vorgestellt, das Sensornetze für verschiedene Anwendungen und für verschiedenen Situationen anpassbar macht, indem es unterschiedliche Kommunikationsparadigmen zur anwendungsgesteuerten, modusbasierten Optimierung der Kommunikation bereitstellt. Innerhalb des Modus-Rahmenwerks werden *a-priori* gewünschte Optimierungsziele der Kommunikation, sogenannte Modi, festgelegt. Ein Modus (bzw. dessen Implementierung) legt das Verhalten eines Knotens innerhalb dieses Modus fest. Eine spezielle Modus-Implementierung kann somit beliebige Ziele verfolgen, z. B. energiesparende Kommunikation (für den Normalbetrieb), schnelle Kommunikation (für Alarmsituationen) oder robuste Kommunikation (für Situationen mit hoher Fehler- oder Ausfallwahrscheinlichkeit). Das Rahmenwerk bietet über eine Schnittstelle die Möglichkeit, zwischen den einzelnen Modi zu wechseln.

Zu Beginn dieses Kapitels wird die Notwendigkeit für ein von der Anwendung steuerbares Kommunikationsverhalten motiviert und der Begriff des Modus definiert. Im darauf folgenden Teil werden die Architektur des Modus-Rahmenwerks vorgestellt, verschiedene Optionen für Modus-Implementierungen vorgeschlagen und die Vorgehensweise beim Modus-Wechsel beschrieben. Im zentralen Abschnitt des Kapitels wird das abstrakte Protokoll beschrieben, das den konsistenten Wechsel zwischen beliebig

vielen und vorher nicht festgelegten Modus-Implementierungen erlaubt. Konkrete Instantiierungen mit relevanten Parametern des Protokolls und exemplarischen Modus-Implementierungen werden im letzten Abschnitt evaluiert.

4.1 Motivation

Wegen der besonderen Rahmenbedingungen bei Sensornetzen steht die Sparsamkeit beim Umgang mit Energie fast immer an erster Stelle. Viele der Routingprotokolle, Betriebssysteme und Algorithmen für Sensornetze werden auf die Energieknappheit der einzelnen Sensorknoten und des Sensornetzes im Gesamten optimiert und entsprechend evaluiert. Während die begrenzte Energie tatsächlich sehr oft den limitierenden Faktor zur Erreichung des eigentlichen Ziels eines Sensornetzes darstellt, ist eine alleinige oder auch nur überwiegende Konzentration auf das Energiesparen nicht jeder Aufgabe angemessen. Vielmehr bestimmt die Aufgabe selbst den limitierenden Faktor, der unter Umständen durchaus nicht Energieknappheit ist. Für lang angelegte Sammlungen von Sensordaten wird eine möglichst lange Laufzeit bei ausreichender Datenlage angestrebt wird, weswegen der bewusste Umgang mit der zur Verfügung stehenden Energie im Mittelpunkt steht. Dagegen spielt die Energie bei einer Interaktion der Sensorknoten mit Menschen, bei Beobachtung von sich innerhalb eines Sensornetzes bewegenden Objekten, bei der Überwachung sensibler Umweltdaten (das Sensornetz als Feuermelder ist ein solches Beispiel) keine Rolle, oder bestenfalls eine nachgeordnete. Vielmehr gilt es bei solchen Aufgaben, die Reaktionszeit des Sensornetzes zu minimieren, die Kommunikation möglichst robust zu gestalten, oder andere aufgabenabhängige Ziele zu erreichen. Für spezielle Randbedingungen werden deswegen auch entsprechende Protokolle für Sensornetze entwickelt, die genau die oben genannten Ziele verfolgen. Offensichtlich ist somit der Anwendungsfall und die spezielle Aufgabe maßgeblich dafür verantwortlich, wie Protokolle und Algorithmen im jeweils betrachteten Sensornetz konzipiert sein müssen.

Wenn die jeweilige Aufgabe die Zielsetzung beziehungsweise den Charakter der Sensornetzprotokolle bestimmt, führt dies zur Frage, ob ein einzelnes Protokoll in jeder Situation den Aufgaben gerecht wird. Tatsächlich gilt genau dies für Sensornetze, die ausschließlich und andauernd eine exakt definierte Anwendung ausführen: Daten sammeln in „nicht ausfallträchtigen“ Umgebungen wird mit energieoptimierenden Protokollen auskommen, Feuer melden in Gebäuden wird auf latenzminimierende Protokolle zurückgreifen, Sensorwerte langfristig zur Verfügung stellen mit hoher Wahrscheinlich-

keit von einzelnen Knotenausfällen werden Protokolle mit hoher Robustheit leisten können.

Ganz anders sieht die Situation für Sensornetze mit wechselnden Aufgaben aus. Hier müssen mit hoher Wahrscheinlichkeit unterschiedliche Ziele berücksichtigt werden, was nicht durch ein einzelnes Protokoll erreicht werden kann. Wechselnde und widersprüchliche Ziele erfordern die Verwendung verschiedener Protokolle, die der jeweiligen Situation angepasst sind.

Diese Betrachtung führt zum Begriff des *Modus*, der die Zielsetzung, Charakter und Umsetzung der verwendeten Mechanismen und Protokolle vereint. In den vorhergehenden Abschnitten wurde von „Situationen“, „Anwendungen“ und „Aufgaben“ gesprochen. Um von der speziellen Betrachtung zu abstrahieren, werden diese umschreibenden Begriffe durch den Begriff *Modus* ersetzt, der in dieser Arbeit folgendermaßen definiert wird:

Definition Modus

> „Ein Modus ist ein Betriebszustand eines Sensornetzes (und damit jedes Knotens). Er legt die verwendeten Protokolle, das Verhalten und die Algorithmen eines Sensornetzes (und damit jedes Knotens) fest.“

Unmittelbar ableitbar aus dieser Definition ist, dass es mehrere „Betriebszustände“, also Modi, im Lebenszyklus eines Sensornetzes geben kann. Konsequenterweise muss es in einem modusbasierten Sensornetz eine Möglichkeit geben, den Modus situationsbedingt zu wechseln. Dieser Wechsel betrifft nach der obigen Definition sowohl den einzelnen Knoten als auch das Sensornetz im Gesamten.

In einem modusbasierten Sensornetz lassen sich folglich zwei Problembereiche identifizieren, die getrennt zu untersuchen sind: zum einen die Protokolle und Mechanismen, die einen vorgegebenen Modus in der geforderten Weise implementieren; zum anderen Protokolle und Mechanismen, die den Wechsel zwischen den Modi umsetzen.

Der erste Problembereich ist sehr gut erforscht, und folglich wurden für sehr viele allgemeine, aber auch spezielle Anforderungen Protokolle entworfen (siehe dazu Abschnitt 2.3.2). Wenig Beachtung wurde jedoch dem Wechsel zwischen verschiedenen Modi geschenkt, bei dem ein Sensornetz sein Verhalten situationsbedingt ändert und an verschiedene Anforderungen anpasst. In dieser Arbeit werden deswegen Modus-Implementierungen als gegeben vorausgesetzt. Soweit notwendig werden beispielhafte Modus-Implementierungen vorgeschlagen und genutzt, die das vorgegebene Ziel des Modus,

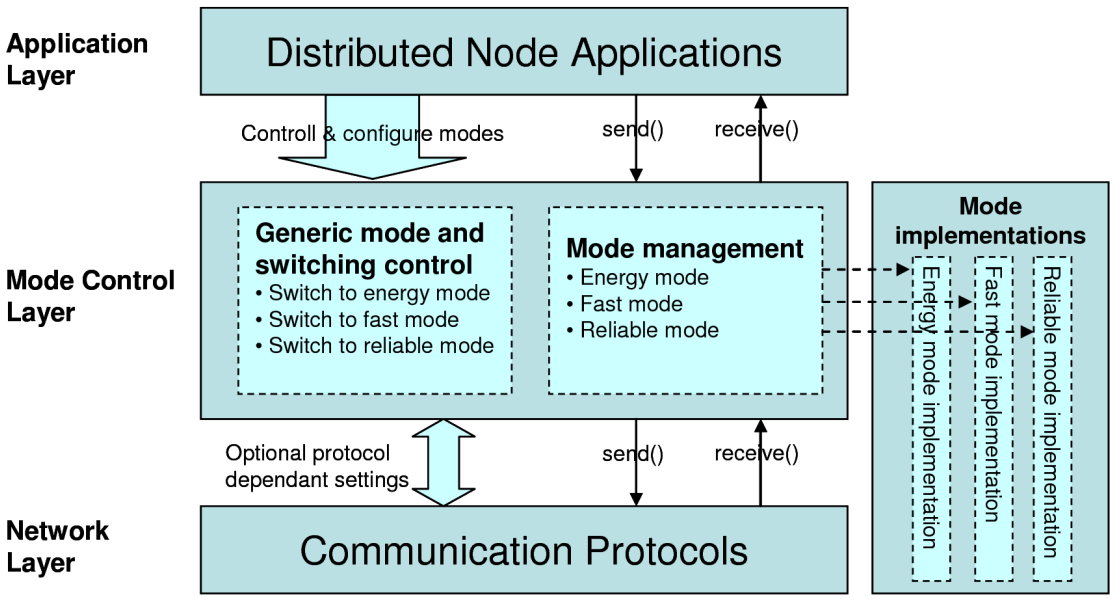

Abbildung 4.1: Architektur des Modus-Rahmenwerks

z. B. Latenzminimierung, umsetzen. Diese Arbeit konzentriert aber im Wesentlichen sich auf den zweiten identifizierten Problembereich, den Wechsel zwischen den Modi.

4.2 Konzept der modusbasierten Kommunikation

Die Architektur des Modus-Rahmenwerks ist in Abb. 4.1 abgebildet. Diese abstrakte Architektur legt die Basis zur Umsetzung der in Abschnitt 4.1 dargelegten Anforderungen, ohne dass eine konkrete Umsetzung vorgegeben wird. Das Architekturmodell legt drei Schichten fest:

- Anwendungsschicht (*Application Layer*)
- Moduskontrollschicht (*Mode Control Layer*)
- Kommunikationsschicht (*Network Layer*)

Application Layer

Vom gesamten Sensornetz aus betrachtet befindet sich in der Anwendungsschicht (*Application Layer*) die eigentliche Anwendung, die über alle teilnehmenden Sensorknoten verteilt ist. Von den einzelnen Sensorknoten aus betrachtet liegt in dieser Schicht der jeweils für sie bestimmte Teil der verteilten Anwendung. Jeder Knoten hat die Möglich-

keit, auf die Modusausgestaltung Einfluss zu nehmen („Controll & configure modes“), und einen Modus zu fordern („send“) bzw. entgegenzunehmen („receive“). Die Möglichkeit zur Einflussnahme auf die Modusausgestaltung hängt dabei im Wesentlichen von den zur Verfügung stehenden Parametern der Modus-Implementierung ab. Beispiele für solche Parameter wären:

- Festlegung und Anpassung von MAC-Parametern,
- Festlegen der Wach- und Schlafendzeiten des drahtlosen Kommunikationsmoduls,
- Anzahl und Festlegung der Kommunikationspartner,
- Wartezeit, bevor Nachrichten weitergeleitet werden,
- Anzahl der Nachrichten, die aggregiert werden.

Wie oben bereits erwähnt, liegt der Fokus dieser Arbeit nicht in der Beschreibung und Ausgestaltung der Modi. Die Liste ist deswegen beispielhaft und nicht umfassend. Die konkreten Parameter hängen ausschließlich von der gewählten Modus-Implementierung ab (Modus-Implementierungen bieten eventuell auch gar keine Einflussmöglichkeit über Parameter).

Network Layer

Ebenso abhängig sind die Einflussmöglichkeiten auf die Protokolle der Kommunikationsschicht (*Network Layer*). In dieser Schicht befinden sich frei wählbare Kommunikationsprotokolle. Voraussetzung für das gewählte Kommunikationsprotokoll ist zum einen die Tauglichkeit für die Modus-Implementierungen, zum anderen eine Möglichkeit zur Information sämtlicher teilnehmender Sensorknoten (Fluten). Die letztgenannte Forderung ist abgeleitet von der Anforderung nach der Möglichkeit zum Modus-Wechsel und wird später in Abschnitt 4.5 vertieft. Um diese Forderung zu erfüllen, müssen folglich alle am Modus-Rahmenwerk teilnehmenden Sensorknoten erreicht (d. h. informiert) werden können. Die abstrakte Modus-Architektur legt dabei aber ausdrücklich nicht die Anzahl und Art der zu verwendenden Kommunikationsprotokolle fest: Die Kommunikationsprotokolle für jede Modus-Implementierung und für die Modus-Wechsel-Implementierungen sind unabhängig und bauen nicht aufeinander auf. Es ist somit nicht notwendig, sich beispielsweise für genau ein Routing-Protokoll (wie z. B. AODV [PBRD03], OLSR [CJ03], DSR [JHM07], etc.) zu entscheiden. Vielmehr steht es jeder Modus-Implementierung sowie jeder Modus-Wechsel-Implementierung frei, beliebige eigene (falls notwendig sogar mehrere) Kommunikationsprotokolle zu nutzen. Insbesondere für das Fluten, das für den Modus-Wechsel benötigt wird, stehen in der Literatur neben dem *klassischen Fluten* zahlreiche alternative Protokolle zur Verfügung (siehe hierzu z. B. adaptives Fluten [LBJ03], clusterbasiertes Fluten [KGV+03], Performancestudie über verschiedene Protokolle zum Fluten in Ad-hoc-Netzen [YGT03]).

4.3 Motivation und Umsetzungsmöglichkeiten der Modi

Kommunikationsstrategie

Modus-Implementierungen dienen laut obiger Definition dazu, dass ein Sensornetz ein gewisses Ziel unter Anteilnahme aller Knoten verfolgt. Zur Verfolgung dieses Ziels können die Knoten sowohl autonom und unabhängig ein gewisses Verhalten annehmen als auch das Verhalten untereinander, also ihre Kommunikationsstrategie, anpassen. Auf der Kommunikationsebene gibt es wiederum zwei Abstraktionsebenen: Die eine umfasst die unmittelbare Kommunikation zweier Kommunikationspartner (Link-Ebene), die andere legt das Verhalten eines gesamten Kommunikationsweges über mehrere Knoten fest (Routing-Ebene). Auch das Verhalten des Kommunikationsweges lässt sich in zwei Kategorien aufteilen: zum einen die klassischen Routing-Protokolle auf Schicht 3 (Netzwerkschicht) des ISO/OSI-Modells; zum anderen Routing-Protokolle, die in der Applikationsschicht implementiert sind und wiederum auf Routing-Möglichkeiten der Schicht 3 aufsetzen. In der folgenden Liste sind die genannten Kategorien zusammengefasst und einige der verschiedenen Implementierungsoptionen aufgezählt. Die Erläuterung der aufgezählten Implementierungsoptionen folgt direkt im Anschluss dieser Liste.

1. Routing (Applikationsschicht)
   a) Applikationsaggregation
   b) Clustering
   c) Acknowledgements
2. Routing (Netzwerkschicht)
   a) Einzelpfad
   b) Multipfad
   c) Netzinterne Aggregation
3. Link
   a) Latenz
   b) Bandbreite
   c) Reichweite
4. Knoten
   a) Schlafmodi
   b) Rechengeschwindigkeit

Die Applikationsaggregation wird genutzt, um in Sensornetzen mehrere Sensorwerte über eine Aggregationsvorschrift zu einem einzigen zusammenzufassen (z. B. Mittelwertbildung). Dies ist immer dann sinnvoll, wenn das eigentliche Ziel der Sensorwerte (die Senke) über mehrere Knoten erreicht werden kann und in der Senke nur der aggregierte Wert von Interesse ist. Mit diesem Verfahren wird die Zahl der Einzelübertragungen gesenkt und somit Energie gespart. Clustering verfolgt in diesem Fall ebenfalls das Ziel, Kommunikationsenergie zu sparen. Dazu werden räumliche Cluster gebildet, die als Kommunikationseinheiten dienen. Jeder Knoten sendet dann seine Daten nicht mehr direkt der Senke, sondern einem ausgewählten Clusterknoten. Dieser schickt die Daten gebündelt weiter (an die Senke oder wieder an einen übergeordneten Cluster). Applikationsspezifische Acknowledgements werden verwendet, um die Übertragung von Daten auf Applikationsebene sicherzustellen. Beim Ausbleiben einer Bestätigung wird das entsprechende Paket erneut gesendet, um bei unzuverlässigen Kommunikationsbedingungen eine zuverlässige Übertragung zu erreichen.

In der Netzwerkschicht kann Energie gespart werden durch eine Übertragung auf nur einem Weg (Einzelpfad), oder entsprechend die Robustheit gegenüber Paketverlusten gesteigert werden durch redundante Übertragung auf mehreren Wegen (Multipfad). Die netzinterne Aggregation dient der Senkung der Energiekosten, indem Daten auf den Routingwegen zusammengefasst (i. d. R. ohne Aggregationsvorschrift einfach aneinandergehängt) werden.

Auf der Link-Schicht kann der Parameter Latenz genutzt werden, um die Weiterleitung zu beschleunigen beziehungsweise zu verzögern, um beispielsweise die netzinterne Aggregation zu unterstützen. Ähnliches gilt für die Parameter Bandbreite und Reichweite, die den Energieverbrauch und die Geschwindigkeit beeinflussen.

Der Sensorknoten selbst kann ebenfalls sein Verhalten an gewisse Situationen anpassen. Hardwareseitig können, soweit vorhanden, die Schlafmodi des Prozessors und des Kommunikationsmoduls genutzt werden, um Energie zu sparen. Eine weitere Möglichkeit stellt die Taktgeschwindigkeit des Prozessors dar. Je nach Anforderung kann diese der gegebenen Aufgabe angepasst und damit die Geschwindigkeit und der Energieverbrauch ausbalanciert werden.

Die genannten Mechanismen und Parameter erheben nicht den Anspruch auf Vollständigkeit. Sie können aber als typische Vertreter in den vier Kategorien angesehen werden und geben insbesondere einen Einblick, auf welche Art in den vier Kategorien das Kommunikationsverhalten beeinflusst werden kann. Eine solche Liste kann aber aus systematischen Gründen nie umfassend sein: Die Mechanismen und Parameter hän-

gen unmittelbar von den verwendeten Protokollen und der Hardware ab und können deswegen stets um weitere Parameter beziehungsweise um neue Mechanismen ergänzt werden. Abhängig vom Modus (also dem Optimierungsziel) kann dann eine Kombination der angegebenen und ggf. weiterer Mechanismen verwendet werden, um diesen zu implementieren. Wenn beispielsweise Energieoptimierung das entscheidende Kriterium des Modus ist, können aus der obigen Liste z. B. die Applikationsaggregation, die netzinterne Aggregation und die Schlafmodi genutzt werden, um Kommunikationskosten und damit Energie zu sparen.

4.4 Exemplarische Umsetzung von Modus-Implementierungen

Obwohl die Anzahl und die Eigenschaften der Modi konzeptuell unerheblich für die modusbasierte Optimierung der Kommunikation sind, ist deren letztendliche Auswahl bei der Realisierung dennoch wichtig für die Brauchbarkeit und Nützlichkeit im späteren Sensornetz. Im Folgenden werden deswegen Modi identifiziert, die einen gewissen Anspruch auf Allgemeingültigkeit haben und sich deswegen auch für eine anschließende Evaluation der modusbasierten Optimierung der Kommunikation eignen.

energiesparend

In den meisten Sensornetzen spielt die Sparsamkeit bezogen auf den Energieverbrauch eine wesentliche Rolle. Dies ist vor allem darin begründet, dass die einzelnen Sensorknoten nur einen begrenzten Vorrat an Energie besitzen und diesen in der Regel auch bis zum Ende des Lebenszyklus des Gesamtsensornetzes nicht regenerieren können. Die Funktion des Sensornetzes ist aber von der Lebensdauer der einzelnen Knoten unmittelbar abhängig. Zum einen liefern die Sensorknoten die essentiellen Daten (Sensorwerte), zum anderen sind sie auch Teil des Kommunikationsnetzes, über das andere Sensorknoten miteinander kommunizieren. Ein Ausfall eines Knotens kann somit sowohl das Fehlen von für die Aufgabe wichtigen Sensordaten (dieses Knotens) bedeuten wie auch die Unterbrechung von Kommunikationspfaden, was in der Folge ebenfalls das Fehlen von Sensordaten (anderer Knoten) bewirkt.

schnell

Neben der Energieoptimierung gibt es aber auch häufig die Anforderung, dass Daten möglichst schnell ihr Ziel erreichen. Dies ist immer dann der Fall, wenn Reaktionen von Aktoren nötig sind, Einstellungen geändert werden, oder das Sensornetz mit einem menschlichen Benutzer interagiert, d. h. Daten angefragt werden. In solchen Situationen muss die Kommunikationsstrategie auf möglichst schnelle Weiterleitung von Daten

optimiert werden, um eine sinnvolle Interaktion zu ermöglichen. Des Weiteren ist eine schnelle Kommunikation immer dann notwendig, wenn zeitsensible Sensordaten gemessen und weitergeleitet werden sollen. Dies ist insbesondere bei Sensornetzanwendungen der Fall, die zur Überwachung und Alarmmeldung dienen. Ist die Aufgabe eines Sensornetzes, Eindringlinge oder Feuer zu melden, muss beim Eintreten des Alarmfalles die Meldung möglichst rasch an die entsprechende Senke geleitet werden.

Eine für die meisten Sensornetze grundlegende Annahme neben der Energieknappheit ist die Unzuverlässigkeit von Kommunikationsverbindungen. Deswegen können viele für Sensornetze entwickelte Kommunikationsprotokolle mit einem (zumindest zeitweiligen und stochastischen) Ausfall von Verbindungen und Knoten umgehen, indem sie reaktiv Alternativ-Routen anlegen. Mit dem modusbasierten Ansatz kann jedoch auch proaktiv einer unzuverlässigen Situation begegnet werden. Erkennt beispielsweise eine Anwendung zur Überwachung eines Waldgebietes einen Brand, kann davon ausgegangen werden, dass viele Knoten ausfallen werden. In einem solchen Fall kann proaktiv auf eine robuste Kommunikation gewechselt werden. Die zentrale Annahme hierbei ist, dass nicht das Kommunikationsprotokoll aufgrund akuter Ausfälle geändert wird, sondern die Sensornetzanwendung schon im Vorhinein von zu erwartenden Ausfällen ausgeht und deswegen auf eine robuste Kommunikation umschaltet.

robust

Die hier vorgestellten drei Optimierungsziele energiesparend, schnell, robust eignen sich für eine Vielzahl von verschiedenen Situation und Anwendungen. Natürlich sind für spezielle Anwendungen auch eine Kombination der Optimierungsziele (z. B. gleichzeitig schnell und robust) vorstellbar, beziehungsweise völlig anders geartete (z. B. Rückverfolgbarkeit der Kommunikationspfade oder verschlüsselte Kommunikation). Wie oben bereits dargelegt, ist die Architektur der modusbasierten Optimierung für solche Erweiterungen offen und prinzipiell für beliebige und beliebig viele Modi geeignet. Im Rahmen dieser Arbeit genügen jedoch diese drei Modi, um das Prinzip der Vorgehensweise darzustellen. Der energiesparende Modus stellt dabei den Standardmodus dar, der immer dann gilt, wenn kein anderer Modus gefordert wird. Die anderen beiden Modi können von Knoten konkurrierend angefordert werden, wobei sich der Modus mit der höheren Priorität netzweit durchsetzt (in den später folgenden Beispielen ist dies der robuste Modus).

Im Folgenden sollen die vorgestellten Modi auf einfache, nachvollziehbare Art und Weise umgesetzt werden. Dabei wird nicht auf eine ausgewiesene Effizienz der einzelnen Modus-Implementierungen Wert gelegt, sondern lediglich eine grundsätzliche Effektivität gefordert, die eine spätere Evaluation der modusbasierten Kommunikation ermöglicht. Ein Modus soll somit in seiner namengebenden Eigenschaft effektiver sein

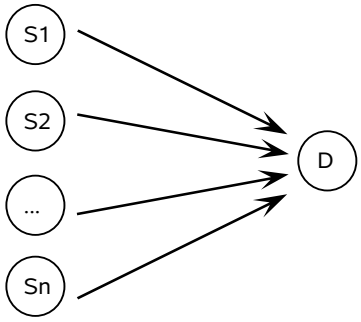

Abbildung 4.2: Kommunikationsgruppe als Grundlage für Modus-Implementierungen

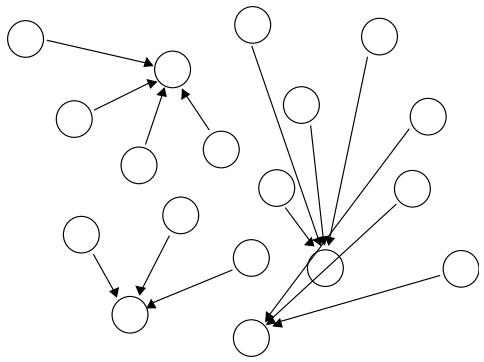

Abbildung 4.3: Sensornetz mit mehreren Kommunikationsgruppen

als die beiden anderen Modi. Diese grundsätzliche Effektivität wird später in Abschnitt 4.9.3 evaluiert und nachgewiesen.

4.4.1 Kommunikationsgruppen als Abstraktion für Kommunikation in Sensornetzen

Für die Implementierung der drei Modi wird angenommen, dass die Kommunikation der zu untersuchenden Sensornetze auf Kommunikationsgruppen basiert. D. h., dass Sensordaten von mehreren Sensorknoten stets zu einer Senke weitergeleitet werden. Abb. 4.2 zeigt eine solche Kommunikationsgruppe. Dabei stehen die mit S (für engl. *source*) bezeichneten Knoten für Datenquellen, der mit D (für engl. *destination*) bezeichnete Knoten für die gemeinsame Senke. Dabei ist zu beachten, dass es mindestens eine Quelle gibt und die Anzahl der Quellen nach oben nicht begrenzt ist.

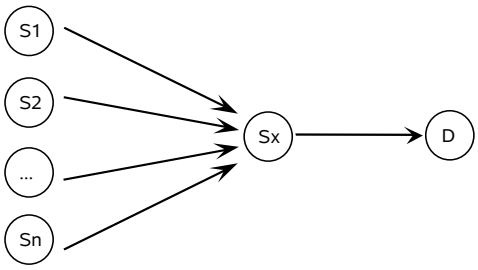

Abbildung 4.4: Umsetzung der Kommunikationsgruppe für energiesparenden Modus

In einem Sensornetz gibt es eine beliebige Anzahl solcher Kommunikationsgruppen. Abb. 4.3 zeigt beispielhaft ein solches in Kommunikationsgruppen aufgeteiltes Sensornetz. Der Übersicht wegen ist in diesem beispielhaften Sensornetz kein Knoten zugleich Quelle und Senke, wobei dies im Allgemeinen natürlich der Fall sein kann.

Grundlage der Entscheidung für eine solche Kommunikationstopologie ist die Erkenntnis, dass viele Sensornetzanwendungen im Wesentlichen auf diese Weise aufgebaut sind. Des Weiteren sind in diesem Modell zwei grundlegende Kommunikationsstrategien in Sensornetzen vereint: Zum einen werden Sensordaten räumlich beieinanderliegender Knoten an eine gemeinsame Senke gesendet, zum anderen kann aber diese Senke räumlich weit entfernt von jenen Sensorknoten liegen.

Als einfaches Beispiel kann die Anfrage nach einer Durchschnittstemperatur dienen: Ein menschlicher Benutzer stellt eine Anfrage nach der durchschnittlichen Temperatur eines entfernten Raumes, in dem sich mehrere Temperatursensorknoten befinden. Der Mittelwert der Temperaturwerte sämtlicher Temperatursensorknoten S muss demnach an denjenigen Knoten D gesendet werden, auf dem der Benutzer die Anfrage gestellt hat. Wie dieser Mittelwert ermittelt wird, ob über netzinterne Aggregation oder Mittelwertbildung erst beim Knoten D, ist für den Benutzer transparent und wird von der Modus-Implementierung bestimmt.

4.4.2 Energiesparender Modus mit Kommunikationsgruppen

Mit der obigen Definition der Kommunikationsgruppe lassen sich auf einfache Weise beispielhafte Implementierungen für die drei Modi energiesparend, schnell, robust umsetzen. Im Falle des energiesparenden Modus ist die energieeffiziente Kommunikation das Optimierungsziel, weswegen hier die Kommunikationsgruppe als Aggregationsgruppe genutzt wird. Ein Mitglied S_x der Kommunikationsgruppe wird als Aggregator

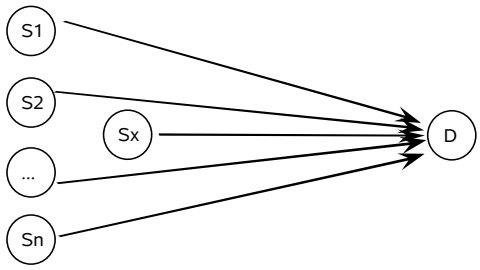

Abbildung 4.5: Umsetzung der Kommunikationsgruppe für schnellen Modus

bestimmt, zu dem alle übrigen Mitglieder ihre Daten senden. Der Aggregator sammelt alle Daten, aggregiert diese (mitsamt seines eigenen Datums) und sendet das Aggregat an die Senke *D*. Abb. 4.4 zeigt die resultierende Topologie. Zu beachten ist, dass der Kommunikationspfad von S_x zu D in der Regel über mehrere andere Knoten führt (Multi-Hop).

4.4.3 Schneller Modus mit Kommunikationsgruppen

Während eine Aggregation auf einem Knoten Datenpakete einspart, wird die Zeitdauer verlängert, bis die Daten bei der Senke ankommen. Der aggregierende Knoten muss eine gewisse Zeit warten, bis alle Daten bei ihm eingetroffen sind und kann erst dann das Gesamtpaket an die Senke senden. Für eine möglichst schnelle Kommunikation ist es hingegen sinnvoll, dass jeder einzelne Knoten seine Daten individuell an die Senke sendet. Dabei wird vorausgesetzt, dass sich die Weiterleitung der einzelnen Daten nicht gegenseitig behindern. Sollte diese der Fall sein, müsste die Modus-Implementierung entsprechend angepasst werden. Für den hier angestrebten Vergleich zwischen verschiedenen Modus-Implementierungen reicht aber die hier vorgestellte Lösung aus. Abb. 4.5 zeigt die Kommunikationspfade für den schnellen Modus. (Der Knoten S_x hat hier keine besondere Bedeutung und wird hier lediglich zum besseren Vergleich mit den anderen Modi ebenfalls aufgeführt.)

4.4.4 Robuster Modus mit Kommunikationsgruppen

Bei einer unzuverlässigen Umgebung wird davon ausgegangen, dass Knoten und Verbindungen ausfallen können. Eine Möglichkeit, dem zu begegnen, ist die Nutzung von redundanten Kommunikationspfaden. Um die Zustellungswahrscheinlichkeit von Daten-

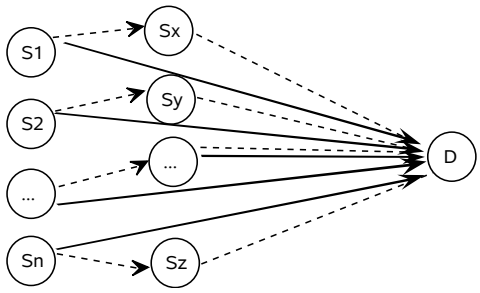

Abbildung 4.6: Umsetzung der Kommunikationsgruppe für robusten Modus

paketen zu erhöhen, wird die Redundanz erhöht, indem das Datenpaket gleichzeitig auf zwei oder mehr verschiedenen Pfaden weitergeleitet wird. Unter Ausnutzung der zuvor definierten Kommunikationsgruppe wird hier der „direkte“ Pfad (bestimmt vom darunterliegenden Schicht-3-Protokoll) mit dem redundanten Pfad über ein Mitglied der Kommunikationsgruppe gewählt. Das Datenpaket wird somit zuerst an ein zufällig ausgewähltes Mitglied der Kommunikationsgruppe gesendet, das anschließend das Paket an den eigentlichen Empfänger (die Senke) weiterleitet. Abb. 4.6 zeigt die abstrahierten Kommunikationspfade für die Implementierung des robusten Modus. Die durchgezogenen Pfeile entsprechen der direkten Kommunikation; die gestrichelten Pfeile entsprechen dem redundanten Weg über ein Gruppenmitglied. Auch hier wird ähnlich dem schnellen Modus vorausgesetzt, dass die redundanten Pfade keinen negativen Einfluss auf die Zustellung der Datenpakete insgesamt haben. Analog zum schnellen Modus gilt aber auch hier, dass die hier gewählte Umsetzung dem Zweck der Illustration (und der späteren Evaluation) genügt.

4.5 Wechsel zwischen Modi

Mit den in Abschnitt 4.4 eingeführten Modi wird eine Nutzung des Sensornetzes für unterschiedliche Situationen ermöglicht. Je nach Situation kann demnach eine passende Kommunikationsstrategie implementiert und genutzt werden. Um die Modi aber effektiv nutzen zu können, muss der Wechsel zwischen diesen möglich sein. Die Ermöglichung dieses Wechsels durch geeignete Protokolle stellt den Kern der hier vorgestellten modusbasierten Optimierung der Kommunikation dar.

4.5.1 Anwendungsspezifische Gründe für einen Moduswechsel

Der in Abschnitt 4.2 beschriebene Ansatz zur modusbasierten Optimierung der Kommunikation stellt die Anwendung in den Mittelpunkt. Nicht die Implementierung des Modus soll auf eine geänderte Situation reagieren, sondern die Anwendung kann das ihr eigene Wissen über die Situation einsetzen, um die richtige Kommunikationsstrategie (also den richtigen Modus) zu wählen.

Dieses Wissen wird desto wichtiger, je umfassender die Auswirkung sein soll. Treten gewisse Besonderheiten lokal begrenzt und zufällig auf (z. B. Störungen der Übertragung zwischen zwei Knoten), sind lokale Aktionen beziehungsweise Reaktionen effizient (z. B. lokale Reparatur von Kommunikationswegen). Auf solche Besonderheiten kann und soll durch die Modus-Implementierungen reagiert werden.

Sind die Besonderheiten dagegen systematisch und global (z. B. voraussehbare Störung der Übertragung zwischen allen Knoten), ist ein im Voraus erfolgender, netzweiter Modus-Wechsel schneller und effizienter, insbesondere wenn auf lokaler (Protokoll-)Ebene die Besonderheit nicht oder erst spät erkannt werden kann.

Ein netzweiter Modus-Wechsel ist auch dann angebracht, wenn die Kommunikation auf fundamentale Weise geändert werden soll (z. B. statt energiesparend soll möglichst schnell kommuniziert werden). In diesen globalen Fällen ist das Wissen der Anwendung entweder schneller („Kommunikation wird schlechter, da ein Feuer ausgebrochen ist.“) oder sogar der einzige Hinweis („Kommunikation soll schnell sein, da es einen Alarm gibt.“).

Bei der modusbasierten Optimierung der Kommunikation wird deswegen davon ausgegangen, dass die Anwendung einen Wechsel des Modus initiiert. Im Sinne der modusbasierten Optimierung der Kommunikation gehört jeder am Modus-Rahmenwerk teilnehmende Knoten zu einer Anwendung. Eine Entscheidung zum Wechsel des Modus muss somit jeder einzelne Knoten auslösen können. Dabei ist es vom Standpunkt des Modus-Rahmenwerks unerheblich, ob die Entscheidung in einer von der Anwendung herrührenden Abstimmung zwischen mehreren Knoten getroffen wird, oder ob ein einzelner Knoten die Entscheidung individuell trifft. Wichtig ist nur, dass letztendlich mindestens ein Knoten den Modus-Wechsel auslöst. Gleichermaßen können aber auch mehrere Knoten gleichzeitig oder zu verschiedenen Zeitpunkten den Modus-Wechsel auslösen.

Als Beispiel kann hier eine Anwendung zur Erkennung eines Feuers mit anschließender Alarmmeldung dienen. In diesem Fall kann ein einzelner Knoten bei Überschreitung eines kritischen Temperaturwertes einen Alarmfall feststellen und den Modus auf „schnelle Kommunikation“ umschalten. Ebenso könnte aber bei einer hohen Streuung der Temperaturmessungen ein gleichzeitiges Überschreiten der kritischen Temperatur einer gewissen Anzahl von Knoten in näherer Umgebung maßgeblich für die Auslösung eines Alarms sein. Die Anwendungsimplementierung würde diese gruppenbasierte Entscheidung in einem dafür geeigneten anwendungsspezifischen Protokoll vornehmen, und derjenige Knoten der Gruppe, der das letztendliche Zutreffen der Alarmbedingung feststellt, würde dann den Modus-Wechsel und den Alarm auslösen.

4.5.2 Notwendigkeit für konsistenten Modus

Die Annahme, dass ein Modus-Wechsel von jedem teilnehmenden Knoten ausgelöst werden kann, führt zu der Situation, dass unter Umständen (beziehungsweise sogar im Normalfall) nicht alle Knoten auf der Anwendungsschicht unmittelbar von diesem Modus-Wechsel erfahren. Am im letzten Abschnitt erwähnten Beispiel des Feueralarms wird dies deutlich. Da nur ein einzelner Knoten, beziehungsweise eine in der Anzahl begrenzte Gruppe von Knoten den Alarmfall auslösen, sind nicht alle Knoten in die Entscheidung einbezogen und erfahren deswegen nichts von der Notwendigkeit eines Modus für „schnelle Kommunikation“. Ein nur teilweiser Wechsel des Modus (d. h. nicht alle Knoten schalten auf „schnelle Kommunikation“) würde aber zu Inkonsistenzen bei der verwendeten Modus-Implementierung führen. Während eine Knoten „schnell“ kommunizieren, verbleiben andere in einer energiesparenden Variante.

Im Fall einer inkonsistenten Sicht auf den aktiven Modus besteht jedoch nicht nur das Problem, dass in einzelnen Bereichen des Sensornetzes unterschiedliche Optimierungsziele bezüglich der Kommunikation gelten und damit bei einer Kommunikation „durch falsche Bereiche“ das Optimierungsziel nicht umgesetzt wird. Vielmehr kann durch eine solche Inkonsistenz die Kommunikation sogar vollständig verhindert werden. Dies ist zum einen der Fall, wenn zur Implementierung der einzelnen Modi inkompatible Protokolle verwendet werden, deren Pakete in Aufbau oder Inhalt vom jeweils anderen Protokoll nicht verarbeitet werden können. Zum anderen wird die Kommunikation verhindert, wenn die unterschiedlichen Modus-Implementierungen von einem festgelegten Verhalten der Nachbarknoten ausgehen (z. B. wartet eine „energiesparende“ aggregierende Gruppe vergeblich auf Daten eines ehemaligen Mitglieds, das durch Modus-Wechsel

auf „schnelle Kommunikation“ nicht an der Gruppe teilnimmt, und verzögert oder verhindert eine Weiterleitung der Daten).

Eine konsistente Sicht auf den aktiven Modus ist somit essentiell notwendig für eine korrekte Funktion. Das Modus-Rahmenwerk muss somit mit geeigneten Mechanismen sicherstellen, dass jeder Knoten über den aktiven Modus informiert wird, wenn die Modus-Implementierungen unabhängig voneinander eingesetzt werden können sollen.

4.5.3 Anforderungen an Modus-Wechsel-Protokolle

Die Modus-Wechsel-Protokolle sind dafür verantwortlich, dass jeder am Modus-Rahmenwerk teilnehmende Knoten den Modus wechselt (Konsistenz). Deswegen ist der Ausgangspunkt eines Modus-Wechsels der Wechselwunsch eines Knotens, dem eine Information aller anderen Knoten folgt, die dann wiederum dem neuen Optimierungsziel (Modus) folgen. Dies bedeutet aber nicht zwingendermaßen, dass alle Knoten auch tatsächlich *aktiv* innerhalb der Modus-Implementierung teilnehmen. Ein Knoten kann auch insofern zur Zielerreichung beitragen, indem er sich passiv verhält. Allerdings ist es auch in diesem Falle notwendig, dass dieser den aktiven Modus kennt. Wenn ein Knoten von einem Modus-Wechsel nicht informiert wird, kann er sich nicht nur undienlich (in Bezug auf das aktuell geltende Optimierungsziel) verhalten sondern sogar kontraproduktiv.

Unter der Annahme einer abhängigen, d. h. auf andere Modi abgestimmten Modus-Implementierung, könnte die Forderung auf Konsistenz insofern abgeschwächt werden, dass in gewissen Toleranzen Inkonsistenzen erlaubt werden. Diese Toleranzen müssten darauf abgestimmt sein, inwieweit die jeweiligen Modus-Implementierungen zusammen harmonieren, ohne das jeweilige Optimierungsziel oder die Funktionsweise zu stören. Dieses Vorgehen widerspräche aber einem unabhängigen Einsatz beliebiger Modus-Implementierungen. Bei einem Einsatz neuer oder geänderter Modus-Implementierungen müssten sämtliche anderen Modus-Implementierungen auf diese Änderungen untersucht und ggf. angepasst werden. Um solche Querbeziehungen zu vermeiden, sollte das Modus-Rahmenwerk darauf verzichten, dass Modus-Implementierungen Eigenschaften anderer Modus-Implementierungen berücksichtigen müssen, um die Funktionalität des Rahmenwerks zu erhalten.

Es ist möglich, dass der Modus-Wechsel sich am vorher geltenden und am gelten sollenden Modus ausrichtet. Dies ist aber nicht zu verwechseln mit der Unabhängigkeit von der Modus-Implementierung. Vielmehr ist es möglich, mehrere Methoden zum Modus-Wechsel zur Verfügung zu stellen. Dabei kann die Art und Weise zu wechseln

sich an den Zielen des Folge-Modus orientieren. So ist beispielsweise ein energiesparender Wechsel angebracht, wenn in einen energiesparenden Modus gewechselt werden soll. Analog gilt natürlich, das ein Wechsel zu einem schnellen Modus ebenfalls schnell und umfänglich (konsistent) erfolgen muss. Gerade im letzteren Fall wäre gegenteiliges Verfahren kontraproduktiv.

Auch wenn ein auf den Folgemodus angepasster Modus-Wechsel unter gewissen Bedingungen vorteilhaft ist, sollen aber die Protokolle zum Modus-Wechsel nicht auf den Modus-Implementierungen selbst aufbauen. Um eine flexible Verwendung des Modus-Rahmenwerks zu gewährleisten, sollen die Modus-Implementierungen unabhängig vom Modus-Wechsel änderbar und austauschbar sein. Im gleichen Maße soll auch die Anzahl der nutzbaren Modi durch die Modus-Wechsel-Protokolle nicht beschränkt sein.

In der folgenden Zusammenstellungen sind die vorangegangen Überlegungen in funktionale und nicht-funktionale Anforderungen kategorisiert und zusammengefasst (zur funktionalen/nicht-funktionalen Einordnung siehe beispielsweise [RR06]). Funktionale Anforderungen an das Modus-Rahmenwerk sind:

1. Möglichkeit zum Wechsel zwischen den Modi;
2. Initiierung des Wechsels durch jeden Knoten möglich;
3. im ganzen Netz muss der gleiche Modus gelten (Konsistenz);

Nicht-funktionale Anforderungen an das Modus-Rahmenwerk sind:

1. freie Wahl der Anwendungen (sie sollen in Anzahl und Art durch das Modus-Rahmenwerk nicht eingeschränkt sein);
2. frei Wahl der Modus-Implementierungen;
3. Unbeschränktheit der Anzahl der Modi;
4. keine Nutzung einer Modus-Implementierung für Modus-Wechsel.

4.6 Prinzipien und Optimierungsmöglichkeiten des Wechsels

Um die funktionalen und nicht-funktionalen Anforderungen umzusetzen, stehen für ein Protokoll zum Modus-Wechsel verschiedenen Mechanismen zur Verfügung. An erster Stelle steht die Notwendigkeit eines Knotens, andere Knoten über den geltenden Modus

informieren zu können (eine sogenannte Modus-Anforderung). Dazu stehen im wesentlichen zwei Möglichkeiten zur Verfügung: Zum einen kann explizit über Nachrichten der geltende Modus verbreitet werden; zum anderen kann die Information implizit durch Auslaufen einer Zeitdauer verbreitet werden. Im letzteren Fall bedeutet dies, dass Modus-Anforderungen zwar vorerst explizit verbreitet werden, diese aber eine Angabe zur Dauer der Modus-Anforderung enthalten. Nach dem Ablaufen dieser Zeit ist keine weitere explizite Benachrichtigung erforderlich, da die Modus-Anforderung „implizit“ ihre Gültigkeit verliert.

Im Gegensatz zur impliziten Informationsverbreitung durch Auslaufen einer Zeitdauer ist es bei der nachrichtenbasierten Informationsverbreitung einer Modus-Anforderung notwendig, dass alle Knoten die Nachricht erhalten, da im gesamten Netz der gleiche Modus gelten soll (siehe funktionale Anforderungen). Deswegen muss hierfür ein Protokoll zum Fluten eines Netzes verwendet werden. Es bietet sich dafür das klassische und einfachste Protokoll zum Fluten an, wo jeder Knoten jede Nachricht genau einmal weitersendet. Es stehen jedoch in der Literatur auch andere Protokolle zum Fluten zur Verfügung, die die Anzahl der zu sendenden Nachrichten reduzieren. Bei der Wahl des Protokolls muss allerdings beachtet werden, dass eine Benachrichtigung aller Knoten garantiert wird. Stochastische Ansätze, bei denen lediglich ein großer Anteil benachrichtigt wird, genügen hier nicht (da sonst die Anforderung, dass im ganzen Netz der gleiche Modus gelten muss, nicht erfüllt werden kann).

In dieser Arbeit wird davon ausgegangen, dass mit dem klassische Fluten alle Knoten erreicht werden. Werden Knoten durch klassisches Fluten nicht erreicht, wird davon ausgegangen, dass diese nicht an der Kommunikation teilnehmen und somit auch nicht wesentlich für einen Modus-Wechsel sind. Für nur temporär ausfallende Knoten gilt diese Voraussetzung allerdings nicht. Um trotzdem die Komplexität für eine ständige Kontrolle der Konsistenzerhaltung aus den Modus-Wechsel-Protokollen herauszuhalten, wird diese Komplexität stattdessen durch folgenden Mechanismus in die Anwendung verlagert: Da die Modi nur eine begrenzte Gültigkeitsdauer besitzen, muss ein Modus durch periodisch wiederkehrende Modus-Anforderungen aufrecht erhalten („aufgefrischt“) werden. Durch diese Bedingung erhalten nur temporär ausgefallene Knoten bei jeder Auffrischung die Möglichkeit, den geltenden Modus einzunehmen und damit die Inkonsistenz zu beenden.

Bei gleichzeitigen, sich widersprechenden Modus-Anforderungen (von verschiedenen Knoten), muss der Konflikt aufgelöst werden können. Aus diesem Grund muss vorgegeben werden, welcher Modus bei sich widersprechenden Modus-Anforderungen gelten soll. Die Modi müssen deswegen wohlgeordnet sein, d. h. jede beliebige Menge von Modi

besitzt genau ein kleinstes beziehungsweise größtes Element. Der nach dieser Wohlordung größte Modus aller aktuellen Modus-Anforderungen soll dann die höchste Priorität besitzen und als der gültige Modus für allen Knoten im Netz gelten.

Existiert zu einem beliebigen Zeitpunkt nur eine Modus-Anforderung, wird durch ein Fluten dieser Modus-Anforderung automatisch ein konsistentes Netz hergestellt, da jeder Knoten über genau diese benachrichtigt ist. Gleichzeitige, sich widersprechende Modus-Anforderungen können jedoch zu Inkonsistenzen führen. Solche Inkonsistenzen können aufgelöst werden, indem Knoten ihre Modus-Anforderung erneut senden (repropagieren), wenn sie von einer widersprüchlichen Modus-Anforderung (explizit oder implizit) informiert werden. Aufgelöst oder sogar vermieden können Inkonsistenzen auch, indem Knoten nicht nur die eigene oder gerade gültige Modus-Anforderung, sondern zusätzlich auch die Modus-Anforderungen anderer Knoten kennen.

Die folgende Liste fasst die oben besprochenen Mechanismen zusammen, die zur Umsetzung eines Protokolls zum Modus-Wechsel zur Verfügung stehen:

1. Informationsverbreitung
   a) Fluten
   b) Time-out
2. Konfliktauflösung
   a) Wohlordnung der Modi
3. Konsistenz
   a) Inkonsistenzen durch Repropagierung auflösen
   b) Inkonsistenzen durch Zustandshaltung vermeiden

4.7 Protokolle zum Modus-Wechsel

Im Folgenden werden Protokolle entwickelt, die die zuvor aufgestellten funktionalen und nicht-funktionalen Anforderungen umsetzen und erfüllen. In Abschnitt 4.6 wurden verschiedene Mechanismen vorgestellt, die für die Protokolle genutzt werden. Zunächst sollen allgemeine Vorgaben vorgestellt werden, bevor in Abschnitt 4.7.2 und Abschnitt 4.7.3 die Protokolle konkretisiert werden.

4.7.1 Generelle Vorgaben und Festsetzungen

Jeder am Modus-Rahmenwerk teilnehmende Knoten muss die Möglichkeit haben, einen Modus-Wechsel zu initiieren. Jeder Knoten kann deswegen mit einer Modus-Anforderung den gewünschten Modus kennzeichnen. Diese Modus-Anforderung übergibt der Knoten an die Moduskontrollschicht. Sie wird als interne Modus-Anforderung D bezeichnet. Eine interne Modus-Anforderung D (für engl. *demand*) beinhaltet den gewünschten Modus D_M und eine Dauer D_V, wie lange der Modus gelten soll:

Interne Modus-Anforderung

$$\text{Interne Modus-Anforderung } D := \{D_M, D_V\}$$

$$\begin{aligned} \text{wobei} \quad D_M &= \text{Modus} \\ D_V &= \text{Gültigkeitsdauer} \end{aligned}$$

Wenn es im Netz konkurrierende Modus-Anforderungen für unterschiedliche Modi gibt, müssen diese Konflikte aufgelöst werden, um einen einheitlichen gültigen Modus feststellen zu können. Diese Konflikte werden aufgelöst, indem jedem Modus eine natürliche Zahl zugeordnet wird. Diese Zahl wird im Folgenden als Kennung bezeichnet. Die Kennung muss dabei in der umgekehrten Richtung auch eindeutig auf den Modus schließen lassen (Bijektivität), was somit eine Wohlordnung der Modi ermöglicht. Die Kennung wird dabei im Vorhinein festgelegt und ändert sich während der Lebenszeit des Sensornetzes nicht. Sind beispielsweise die drei Modi „energiesparend“, „schnell“ und „robust“ vorgesehen, könnte die Zuordnung folgendermaßen aussehen:

Wohlordnung durch Kennung

- Kennung 0: Modus „energiesparend“,
- Kennung 1: Modus „schnell“,
- Kennung 2: Modus „robust“.

Die Kennung wird dazu genutzt, Konflikte bei mehreren Modus-Anforderungen aufzulösen: derjenige Modus mit der größeren Kennung ist der gültige.

Verallgemeinert auf alle aktuellen Modus-Anforderungen (als Menge $\mathcal{D}$ bezeichnet) soll der Modus N_M gelten. Dieser wird als Netz-Modus bezeichnet. Er besitzt in der Menge aller Modus-Anforderungen (mit $D(x)_M$ als Modus-Anforderung des Knotens x) die größte Kennung. In einem konsistenten und korrekten Netz-Zustand sollen alle Knoten den Modus N_M haben. Um jederzeit einen gültigen Modus bestimmen zu können, ist der Modus mit der kleinsten Kennung der Standard-Modus M_0. Dieser wird gültig, wenn keine aktuellen Modus-Anforderungen bestehen. Bei gegebener Menge $\mathcal{D}$ aller Modus-Anforderungen gilt demnach formal:

Standard-Modus

Netz-Modus

$$N_M = \begin{cases} D(x)_M | D(x)_M \geq D(y)_M \forall D(x)_M, D(y)_M \in \mathcal{D} & \text{, falls } \mathcal{D} \neq \emptyset, \\ M_0 & \text{, sonst.} \end{cases}$$

wobei $D(x)_M, D(y)_M$ = Modusanforderung eines Knotens x beziehungsweise y
$\mathcal{D}$ = Menge aller Modus-Anforderungen
M_0 = Standard-Modus

Unabhängig von einer eventuell eigenen Modus-Anforderung D hat jeder Knoten einen Status S. Dieser Status repräsentiert den eigentlichen Modus, in dem sich der Knoten befindet. Die Modus-Anforderung D kann beispielsweise eine Anforderung nach einem niedriger prioren Modus sein, der erst dann aktiv wird, wenn die Gültigkeitsdauer des Status-Modus ausläuft. Der Status S ist gekennzeichnet durch den Modus S_M, der für diesen Knoten gilt, und dessen Gültigkeitsdauer S_V:

Status

$$\text{Status } S := \{S_M, S_V\}$$

wobei S_M = Modus
S_V = Gültigkeitsdauer

Um andere Knoten über Modus-Anforderungen zu informieren, können Knoten Modus-Anforderungspakete senden beziehungsweise empfangen. Ein Modus-Anforderungspaket P enthält den angeforderten Modus P_M und dessen Gültigkeitsdauer P_V:

Modus-Anforderungspaket

$$\text{Paket } P := \{P_M, P_V\}$$

wobei P_M = Modus
P_V = Gültigkeitsdauer

In Abb. 4.7 ist der Automat dargestellt, der den generellen Ablauf der in den folgenden Abschnitten vorgestellten Modus-Wechsel-Protokolle zeigt. Jeder am Modus-Rahmenwerk teilnehmende Knoten implementiert diesen Automaten. Er besteht aus vier Zuständen:

1. Warten,
2. Analyse,
3. Senden,
4. Kontrolle.

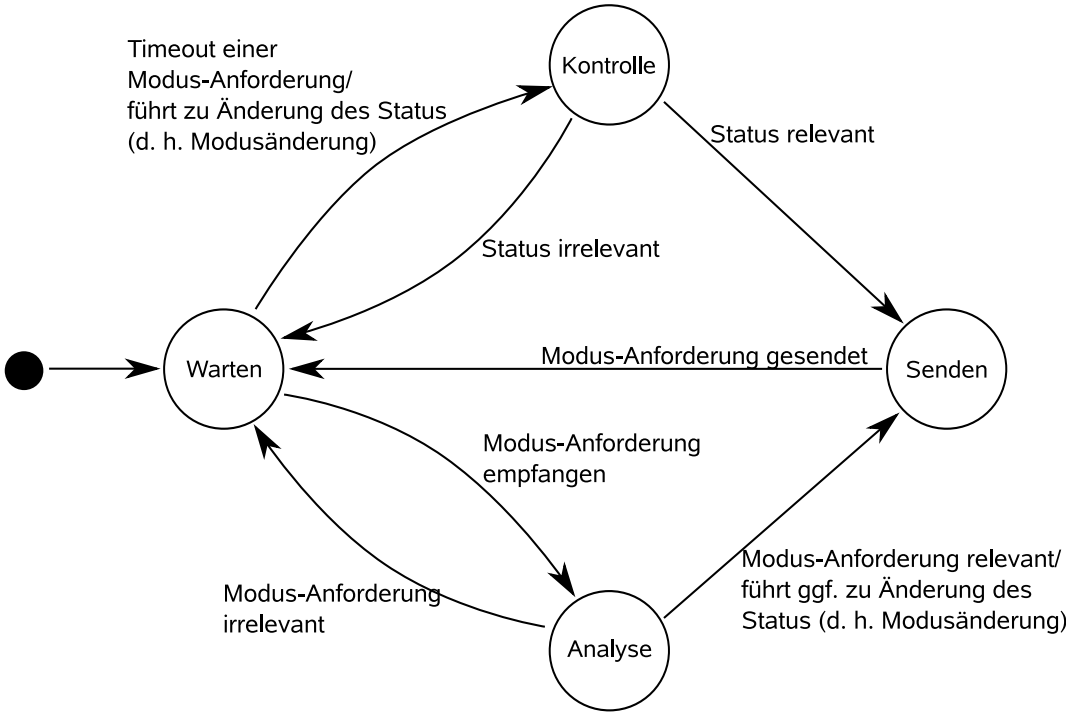

Abbildung 4.7: Genereller Automat für Modus-Wechsel-Protokoll

Der Automat beginnt im Startzustand „Warten“. Von diesem Zustand führen zwei Ereignisse zu einem Zustandsübergang: die Ankunft einer Modus-Anforderung („Modusanforderung empfangen“) und das Auslaufen einer Gültigkeit einer Modus-Anforderung („Timeout einer Modus-Anforderung “). Im Falle des Auslaufens einer Gültigkeit wird ggf. der Status des Knotens angepasst, d. h. der Modus geändert, wenn eine im Cache gehaltene Modus-Anforderung einen neuen Modus ergibt. Ist kein Cache vorhanden oder ist der Cache leer, wird definitionsgemäß der Standard-Modus mit der Kennung 0 eingenommen. Soll eine Modus-Anforderung weiterbestehen, muss der Knoten diese vor Ablaufen der Gültigkeitsdauer auffrischen, indem er eine neue, gleiche Modus-Anforderung setzt.

Analyse einer Modus-Anforderung

Eine Modus-Anforderung kann sowohl von anderen Knoten über ein Modus-Anforderungspaket P empfangen werden als auch eine interne Modus-Anforderung D darstellen. Beides wird als „Modusanforderung empfangen“ behandelt und führt zum Zustand „Analyse“. Hier wird analysiert, ob die neue Modus-Anforderung *relevant* ist. Entschei-

dend für die Relevanz einer Modus-Anforderung ist der momentane Status S eines Knotens, die Kennung des angeforderten Modus und das eventuell vorhandene beziehungsweise ableitbare Wissen über andere Knoten. Eine Modus-Anforderung ist genau dann relevant, wenn gilt:

1. der Status S des Knotens ändert sich und/oder
2. die Modus-Anforderung ist für andere Knoten relevant.

Ob die Modus-Anforderung für andere Knoten relevant ist, wird nach folgenden Kriterien entschieden:

Im Status S ist definitionsgemäß der für die Knoten geltende Modus abgelegt. Wenn sich der Status S ändert, wird davon ausgegangen, dass auch andere Knoten ihren Status ändern müssen, weswegen die Modus-Anforderung dann relevant ist. Wenn sich der Status nicht ändert, ist eine Modus-Anforderung genau dann relevant, wenn sie zu späteren Zeitpunkten den Status beeinflusst. Dies ist z. B. der Fall, wenn der aktive Status-Modus S_M mit der Dauer S_V ausläuft. Nicht relevant wäre eine Modus-Anforderung beispielsweise, wenn sich schon eine höherpriore Modus-Anforderung im Cache befindet. Die Entscheidung, wie letzteres Kriterium der Relevanz angewendet wird, richtet sich nach dem verwendeten Protokoll.

Wird die Modus-Anforderung nach der Analyse als irrelevant erkannt („Modusanforderung irrelevant“), wechselt der Zustand wieder zu „Warten“. Ist die Modus-Anforderung relevant, wechselt der Zustand zu „Senden“.

Senden einer Modus-Anforderung

Im Zustand „Senden“ wird ein Modus-Anforderungspaket an andere Knoten gesendet. Der Inhalt des Modus-Anforderungspakets P hängt dabei vom konkreten Modus-Wechsel-Protokoll ab. Zusätzlich zum oben genannten Modus P_M und der Dauer P_V können beispielsweise der ursprüngliche Initiator dieser Modus-Anforderung oder eine Sequenznummer enthalten sein. Um Duplikate beim Fluten zu erkennen, wird in allen nachfolgend beschriebenen Modus-Wechsel-Protokollen eine Sequenznummer verwendet. Nachdem das Modus-Anforderungspaket gesendet worden ist, wechselt der Zustand wieder in „Warten“.

Kontrolle des Status

Läuft im Zustand „Warten“ die Gültigkeitsdauer einer Modus-Anforderung aus („Timeout einer Modus-Anforderung “), wechselt der Zustand in „Kontrolle“. Ähnlich zum Zustand „Analyse“ wird hier kontrolliert, ob das Auslaufen der Dauer Auswirkungen auf den aktuellen Status S des Knotens hat. Wird der Status geändert und ist die Statusänderung aus der lokalen Sicht des Knotens relevant für andere Knoten („Status relevant“), wird in den Zustand „Senden“ gewechselt, wo die Statusänderung als neue Modus-Anforderung gesendet wird. Wird der Status nicht geändert oder ist die Statusänderung irrelevant für andere Knoten (da diese z. B. durch vorhergehende Modus-Anforderungspakete genügend Information besitzen), wechselt der Zustand mit dem Zustandsübergang „Status irrelevant“ ohne weitere Aktion wieder in den Zustand „Warten“.

Die Kontrolle, ob das Auslaufen der Gültigkeitsdauer Auswirkungen auf den aktuellen Status S hat, ist beim Zustand „Analyse“ Teil der dortigen Zustandsaktionen. In den folgenden Modus-Wechsel-Protokollen wird dies genutzt, indem im Zustand „Analyse“ bei identischen Teilen auf die Überprüfungen und Aktionen des Zustands „Kontrolle“ verwiesen wird.

4.7.2 Modus-Wechsel mit Zeroknowledge

In Abb. 4.7 wurde ein Automat vorgestellt, der die Abläufe für alle in dieser Arbeit entwickelten Modus-Wechsel-Protokolle vereint. Im Folgenden wird dieser allgemeingültige Automat für das spezielle Modus-Wechsel-Protokoll *Zeroknowledge* konkretisiert. Während der prinzipielle Ablauf des Protokolls dem Automaten von Abb. 4.7 folgt und auch die generellen Festsetzungen von Abschnitt 4.7.1 gelten, werden die einzelnen Vorgaben zu Status, Modus-Anforderungspaket, etc. um spezifische Informationen erweitert.

Das Modus-Wechsel-Protokoll *Zeroknowledge* soll die Vorgaben der funktionalen Anforderungen möglichst einfach umsetzen, d. h. insbesondere ohne die Speicherung von Zuständen beziehungsweise Modus-Anforderungen anderer Knoten auskommen. Die zentrale Idee ist es, einen konsistenten und korrekten Netz-Modus N_M dadurch herzustellen, indem jede relevante Änderung des Modus an alle Knoten kommuniziert wird, ohne dabei zusätzliche Modus-Anforderungen zwischenzuspeichern. Zur vollständigen Verbreitung der Information einer Modus-Anforderung wird deswegen Fluten verwendet, wobei mithilfe einer Sequenznummer garantiert wird, dass Modus-Anfor-

derungspakete höchstens einmal weitergesendet werden. Durch dieses Verfahren wird jedoch nicht garantiert, dass jeder Knoten das Modus-Anforderungspaket tatsächlich empfängt. Wie bereits im Abschnitt 4.6 zu den Prinzipien des Modus-Wechsels diskutiert, wird die Komplexität der fortwährenden Konsistenzerhaltung nicht in den Modus-Wechsel-Protokollen selbst aufgenommen. Stattdessen haben Modus-Anforderungen nur eine begrenzte Gültigkeit und müssen periodisch von der Anwendung aufgefrischt werden. Dadurch erhalten Knoten, die nur temporär ausfallen und ein Modus-Anforderungspaket nicht erhalten, bei jeder späteren „Auffrischung“ die Möglichkeit, den geltenden Modus zu erhalten.

Der Status S wird somit um eine Sequenznummer S_S erweitert; das Modus-Anforderungspaket P enthält dementsprechend zusätzlich eine Sequenznummer P_S. Die Handhabung der Sequenznummern und weitere Details des Protokolls werden im Anschluss an die folgenden, zusammenfassenden Auflistungen erläutert. Für die interne Modus-Anforderung D ergeben sich keine Änderungen. Für das Modus-Wechsel-Protokoll *Zeroknowledge* gelten folgende Vorgaben:

Interne Modus-Anforderung D

1. D_M: intern angeforderter Modus
2. D_V: Gültigkeitsdauer des intern angeforderten Modus

Status S eines Knotens

1. S_M: Aktiver Modus des Knotens
2. S_V: Gültigkeitsdauer des aktiven Modus
3. S_S: Sequenznummer des letzten empfangenen Modus-Anforderungspakets

Modus-Anforderungspaket P

1. P_M: Angeforderter Modus
2. P_V: Gültigkeitsdauer des angeforderten Modus
3. P_S: Sequenznummer

Im Folgenden werden die Bestandteile des generellen Automaten von Abb. 4.7 für das Modus-Wechsel-Protokoll *Zeroknowledge* konkretisiert. Nach einer zusammenfassenden

Auflistung der einzelnen Vorgehensweisen bei Analyse, Kontrolle, Senden, etc. wird das Vorgehen jeweils näher erläutert.

Neue interne Modus-Anforderung

Neues Paket mit $P_M := D_M, P_V := D_V, P_S := S_S + 1$

Analyse eines Modus-Anforderungspakets

Bedingung		Aktion
(1)	$P_S < S_S$	Verwerfen des Pakets
(2.1)	$P_S = S_S \wedge P_M \leq S_M$	Verwerfen des Pakets
(2.2)	$P_S = S_S \wedge P_M > S_M$	Statusänderung ($S_M := P_M$, $S_V := P_V$), Paket P' mit $P'_M := P_M, P'_S := P_S, P'_V = P_V -$ Zeitverzögerung, Senden von P', falls $P'_V > 0$
(3)	$P_S > S_S$	Statusänderung ($S_M := P_M, S_V := P_V, S_S := P_S$), Kontrolle, Senden von P' (Belegung siehe (2.2)), falls in der Kontrolle kein neues Paket gesendet wurde

Auslaufen einer Gültigkeitsdauer (Timeout)

Bedingung	Aktion
Interne Modus-Anforderung D_V läuft aus	Löschen der internen Modus-Anforderung, keine weiteren Maßnahmen
Status S_V läuft aus	Kontrolle

Kontrolle

Bedingung	Aktion
$S_M \geq D_M$	keine Maßnahme
$S_M < D_M \wedge D_V < V_{min}$	keine Maßnahme
$S_M < D_M \wedge D_V \geq V_{min}$	neues Paket P senden mit $P_M := D_M$, $P_V := D_V$, $P_S := S_S + 1$

Erläuterungen zum Modus-Wechsel-Protokoll *Zeroknowledge*

Eine *neue, interne Modus-Anforderung* wird gemäß dem generellen Automaten von Abb. 4.7 gleich behandelt wie die Ankunft eines Modus-Anforderungspakets. Dazu wird ein neues Paket „fingiert“, das mit den entsprechenden Daten der Modus-Anforderung D befüllt wird. Als Sequenznummer wird die um eins erhöhte Sequenznummer gewählt, die auf dem entsprechenden Knoten unter S_S gespeichert ist. Durch die erhöhte Sequenznummer wird in der *Analyse* eine Überprüfung der neuen, internen Modus-Anforderung gewährleistet.

Die *Analyse eines Modus-Anforderungspakets* prüft bei der Ankunft eines Modus-Anforderungspakets anhand der Sequenznummer und des Modus, ob die Information relevant ist. Ist die Sequenznummer P_S des Pakets kleiner als die im Status enthaltene Sequenznummer S_S wird das Paket verworfen. Es wird dann davon ausgegangen, dass das Paket entweder bereits empfangen und verarbeitet wurde, oder dass es sich um ein veraltetes Paket handelt. Genauso wird ein Paket behandelt, dessen Sequenznummer gleich der Sequenznummer des Status ist, und dessen Modus P_M kleiner beziehungsweise gleich dem Modus S_M im Status ist. Für den Vergleich wird die Wohlordnung der Modi (durch Zuordnung einer Kennung) verwendet. Hat ein Paket die gleiche Sequenznummer aber einen größeren Modus, wird der Status entsprechend angepasst (der Status-Modus S_M wird zu Paket-Modus P_M) und das Paket P' generiert. Das Paket P' hat dabei den gleichen Inhalt, was Modus P_M und Sequenznummer P_S anbelangt. Die Dauer P'_V wird entsprechend einer eventuellen Zeitverzögerung beim Senden korrigiert (z. B. um die Wartezeit, wenn der Sendekanal belegt ist). Wird ein Paket mit größerer Sequenznummer erhalten, wird zunächst ohne Prüfung des aktuellen Status der Status angepasst (mit Modus, Dauer und Sequenznummer). Die anschließende „Kontrolle“ dient dazu, zu ermitteln, ob die interne Modus-Anforderung eine höhere Kennung besitzt (siehe dazu unten bei „Kontrolle“). Nur für den Fall, das bei der Kontrolle keine

interne Modus-Anforderung gesendet wird, wird das Paket P' mit gleichem Inhalt weitergesendet (wieder mit entsprechender Korrektur der Dauer P'_V).

Läuft die Gültigkeitsdauer eines Modus aus (Timeout), wird anhand des Algorithmus für die Kontrolle überprüft, ob der sich ergebende neue Status ein Senden der internen Modus-Anforderung erfordert. Dies gilt allerdings nur für das Auslaufen des Status-Modus, nicht aber für das Auslaufen der internen Modus-Anforderung. Diese ist schon durch den Status repräsentiert: Ist die interne Modus-Anforderung maßgeblich für den Status (d. h. Modus und Dauer sind bei beiden gleich), führt der Timeout beim Status automatisch zu einer *Kontrolle*, wenn auch die interne Modus-Anforderung ausläuft; ist die interne Modus-Anforderung *nicht* maßgeblich für den Status (d. h. deren Kennung ist kleiner), hat das Auslaufen der Gültigkeit auch keinen Einfluss auf den Status dieses Knotens.

Die *Kontrolle* übernimmt die Prüfung, ob der aktuelle Status S und eine eventuell vorhandene interne Modus-Anforderung D das Senden eines neuen Modus-Anforderungspakets erforderlich machen. Ist der Status-Modus größer oder gleich dem Modus der internen Modus-Anforderung, ist keine Maßnahme erforderlich. Ist dagegen der Status-Modus kleiner (d. h. weniger prior im Sinne der Wohlordnung), wird ein neues Modus-Anforderungspaket mit dem Inhalt der internen Modus-Anforderung und einer um eins erhöhten Sequenznummer geschickt. Zu beachten ist dabei lediglich, dass eine gewisse Mindestdauer V_{min} der Gültigkeit eingehalten wird. Die Mindestdauer wird auf einen Wert gesetzt, der in etwa der Zeitdauer für das vollständige Fluten entspricht, um zu garantieren, dass Modus-Anforderungen bei allen Knoten eine gewisse Gültigkeitsdauer besitzen. Diese Zeitdauer muss für das jeweilige Sensornetz berechnet, experimentell bestimmt, oder geschätzt werden.

4.7.3 Modus-Wechsel mit Multiknowledge

Im Folgenden wird das Modus-Wechsel-Protokoll *Multiknowledge* vorgestellt. Das zuvor vorgestellte Modus-Wechsel-Protokoll *Zeroknowledge* zeichnet sich dadurch aus, dass außer des aktiven Modus keine Informationen über Modus-Anforderungen anderer Knoten gehalten werden. Im Modus-Wechsel-Protokoll *Multiknowledge* wird hier ein Cache für zusätzliche Modus-Anforderungen eingeführt. Dieser dient dazu, Konflikte sich widersprechender Modus-Anforderungen schneller als bei *Zeroknowledge* aufzulösen. Auch bei diesem Protokoll gilt der in Abb. 4.7 vorgestellte prinzipielle Ablauf, und die generellen Festsetzungen von Abschnitt 4.7.1. Die einzelnen Vorgaben zu Status, Modus-

Anforderungspaket, etc. werden wie im vorhergehenden Abschnitt um spezifische Informationen erweitert.

Die zentrale Idee des Modus-Wechsel-Protokolls *Multiknowledge* ist es, einen konsistenten und korrekten Netz-Modus N_M dadurch herzustellen, indem Modus-Anforderungen in einem Cache soweit wie möglich zwischengespeichert werden. Das Protokoll ist dabei so generell angelegt, dass die Größe des Caches (gemessen in der Anzahl der gespeicherten Modus-Anforderungen) für alle Cache-Größen ≥ 1 frei wählbar ist. Im Gegensatz zum Modus-Wechsel-Protokoll *Zeroknowledge* erlaubt das Modus-Wechsel-Protokoll *Multiknowledge nicht*, dass ein Knoten vor Ablauf der Gültigkeitsdauer einer internen Modus-Anforderung einen *kleineren* internen Modus anfordert. Lediglich die Anforderung eines *größeren* Modus wird erlaubt. Der Grund hierfür ist die Möglichkeit zu Inkonsistenzen der Caches. Um einen kleineren internen Modus anzufordern, muss ein Knoten auf den Ablauf der Gültigkeitsdauer der noch laufenden internen Modus-Anforderung warten.

Zur vollständigen Verbreitung der Information zu einer Modus-Anforderung wird auch hier Fluten verwendet.

Der Status S wird um eine lokale Sequenznummer erweitert. Die lokale Sequenznummer S_{SL} dient dazu, verschiedene Modus-Anforderungen desselben Knotens unterscheiden zu können. Die lokale Sequenznummer führt jeder Knoten individuell. Sie wird ausschließlich bei jedem neuen, selbst initiierten Modus-Anforderungspaket erhöht und wird von keiner anderen Sequenznummer beeinflusst. Mithilfe der lokalen Sequenznummer wird garantiert, dass eine Modus-Anforderung eines Knotens von jedem anderen Knoten höchstens einmal weitergesendet wird und damit das Fluten konvergiert.

Das Modus-Anforderungspaket P enthält analog die Sequenznummer P_{SL}. Zusätzlich enthält das Paket den Initiator P_I der Modus-Anforderung und den aktiven Modus des Initiators. Für die interne Modus-Anforderung D ergeben sich keine Änderungen.

Der Cache besitzt eine im Vorhinein festgelegte Größe $n \geq 1$ und bietet Platz für n verschiedene Modus-Anforderungen. Ein einzelner Cache-Eintrag $C(x)$ (wobei $x \in [1; n]$) besteht aus dem Initiator $C(x)_I$ der Modus-Anforderung, dem Modus $C(x)_M$, der Gültigkeitsdauer $C(x)_V$ und der Sequenznummer $C(x)_S$.

Für das Modus-Wechsel-Protokoll *Multiknowledge* gelten folgende Vorgaben:

Status S

1. S_I: Initiator des aktiven Modus
2. S_M: Aktiver Modus
3. S_V: Gültigkeitsdauer des aktiven Modus
4. S_{SL}: Lokale Sequenznummer

Cache C

1. $C(1)_I$: Initiator der Modus-Anforderung
   $C(1)_M$: angeforderter Modus
   $C(1)_V$: Gültigkeitsdauer des angeforderten Modus
   $C(1)_S$: Sequenznummer der letzten Modus-Anforderung des Cache-Eintrags 1
2. $C(2)_I$: Initiator der Modus-Anforderung
   $C(2)_M$: angeforderter Modus
   $C(2)_V$: Gültigkeitsdauer des angeforderten Modus
   $C(2)_S$: Sequenznummer der letzten Modus-Anforderung des Cache-Eintrags 2
   $\vdots$

n. $C(n)_I$: Initiator der Modus-Anforderung
   $C(n)_M$: angeforderter Modus
   $C(n)_V$: Gültigkeitsdauer des angeforderten Modus
   $C(n)_S$: Sequenznummer der letzten Modus-Anforderung des Cache-Eintrags n

Interne Modus-Anforderung D

1. D_M: intern angeforderter Modus
2. D_V: Gültigkeitsdauer des intern angeforderten Modus

Modus-Anforderungspaket P

1. P_I: Initiator dieser Modus-Anforderung
2. P_M: Angeforderter Modus
3. P_V: Gültigkeitsdauer des angeforderten Modus
4. P_{SL}: Sequenznummer dieser Modus-Anforderung

Im Folgenden werden die Bestandteile des generellen Automaten von Abb. 4.7 für das Modus-Wechsel-Protokoll *Multiknowledge* konkretisiert. Nach einer zusammenfassenden Auflistung der einzelnen Vorgehensweisen bei Analyse, Kontrolle, Senden, etc. wird das Vorgehen jeweils näher erläutert.

Interne Modus-Anforderung

- Neue interne Modus-Anforderung: neues Paket P mit
  $P_I :=$ dieser Knoten,
  $P_M := D_M$,
  $P_V := D_V$,
  $P_{SL} := S_{SL} + 1$.
- Geänderte interne Modus-Anforderung:
  1. $D_{M_{neu}} \leq D_{M_{alt}}$: nicht erlaubt,
  2. $D_{M_{neu}} > D_{M_{alt}}$: neues Paket mit
     $P_I :=$ dieser Knoten,
     $P_M := D_M$,
     $P_V := D_V$,
     $P_{SL} := S_{SL} + 1$.

Weitersenden einer Modus-Anforderung P

- Paket P' mit
  $P'_I := P_I$,
  $P'_M := P_M$,
  $P'_V := P_V -$Zeitverzögerung,
  $P'_{SL} := P_{SL}$.
- P' senden, falls $P'_V > V_{min}$.

Analyse eines Modus-Anforderungspakets

Bedingung			Aktion
(1)	$\exists x : P_I = C(x)_I \wedge P_{SL} > C(x)_S$		Aufnahme in den Cache, Kontrolle, Weitersenden.
(2)	$\forall x : P_I \neq C(x)_I$		
(2.1)	(2)$\wedge$	$(\exists y : C(y) = \oslash)$	Aufnahme in den Cache, Kontrolle, Weitersenden.
(2.2)	(2) $\wedge \neg$(2.1)$\wedge$	$(\exists y : C(y)_M < P_M)$	Aufnahme in den Cache, Kontrolle, Weitersenden.
(2.3)	(2) $\wedge \neg$(2.1) $\wedge \neg$(2.2)$\wedge$	$(\exists y : C(y)_M = P_M \wedge C(y)_V < P_V)$	Aufnahme in den Cache, Kontrolle, Weitersenden.
(3)	$\neg$(1) $\wedge \neg$(2) $\wedge \neg$(2.1) $\wedge \neg$(2.2) $\wedge \neg$(2.3)		Verwerfen des Pakets.

Aufnahme in den Cache

Der Inhalt des Pakets wird in den Cache übernommen. Falls der Initiator schon einen Cache-Platz besetzt wird dieser Platz aktualisiert. Andernfalls wird entweder ein leerer Cache-Platz belegt (falls vorhanden), der Cache-Platz mit dem kleinsten Modus, oder der Cache-Platz mit dem kleinsten Modus und der kleinsten Gültigkeitsdauer aktualisiert.

Auslaufen einer Gültigkeitsdauer

Bedingung	Aktion
Interne Modus-Anforderung endet	Löschen der internen Modus-Anforderung, keine weiteren Maßnahmen.
Cache-Eintrag läuft aus	Löschen des verantwortlichen Cache-Eintrags, keine weiteren Maßnahmen.
Aktueller Modus läuft aus	Löschen des verantwortlichen Cache-Eintrags, Kontrolle.

Kontrolle

Der Status, der in der folgenden Tabelle zugrunde gelegt wird, ergibt sich aus dem Inhalt des Caches. Derjenige Cache-Eintrag mit dem größten Modus (bei mehreren Kandidaten gilt noch zusätzlich die längste Gültigkeitsdauer) bestimmt den aktuellen Status.

Bedingung	Aktion
$S_M \geq D_M$	keine Maßnahme.
$S_M < D_M \wedge D_V < D_{V_{min}}$	keine Maßnahme.
$S_M < D_M \wedge D_V \geq D_{V_{min}}$	neues Paket P mit $P_I :=$ dieser Knoten, $P_M := D_M$, $P_V := D_V$, $P_{SL} := S_{SL} + 1$.

Erläuterungen zum Modus-Wechsel-Protokoll *Multiknowledge*

Eine *neue, interne Modus-Anforderung* wird gemäß dem generellen Automaten von Abb. 4.7 gleich behandelt wie die Ankunft eines Modus-Anforderungspakets. Bei einer neuen, internen Modus-Anforderung wird deswegen ein solches Paket erstellt und mit den entsprechenden Daten der Modus-Anforderung befüllt. Als lokale Sequenznummer wird die um eins erhöhte Sequenznummer des Status gewählt.

Die *Analyse eines Modus-Anforderungspakets* prüft mithilfe der Cache-Einträge und des aktuellen Status, ob die Information des Modus-Anforderungspakets P relevant ist. Ist der Initiator der P_I Modus-Anforderung bereits im Cache (siehe Bedingung 1) und ist die lokale Sequenznummer P_{SL} größer als die des entsprechenden Cache-Eintrags, wird genau dieser Cache-Eintrag mit den Informationen des Pakets P aktualisiert. D. h. der Modus, die Dauer und die Sequenznummer werden den Paketinformationen entsprechend gesetzt, während der Initiator gleich bleibt. Ist die Sequenznummer kleiner oder gleich, wird das Paket verworfen.

Ist der Initiator P_I der Modus-Anforderung nicht im Cache (siehe Bedingung 2), wird nach einem geeigneten Platz im Cache gesucht (Bedingungen 2.1 bis 2.3). Ist ein Cache-Platz frei (Bedingung 2.1), wird dieser Cache-Platz mit den Informationen des Pakets P belegt. Gibt es einen Cache-Eintrag y, dessen Modus $C(y)_M$ kleiner ist als P_M (siehe Bedingung 2.2), wird dieser Cache-Eintrag durch die Informationen des

Pakets P ersetzt. Gibt es einen Cache-Eintrag, dessen Modus gleich dem des Pakets ist, die Gültigkeitsdauer aber länger ist (siehe Bedingung 2.3), wird dieser Cache-Eintrag durch die Informationen des Pakets P ersetzt. In allen anderen Fällen wird das Paket verworfen, und die Informationen werden nicht in den Cache übernommen. Im Falle einer Übernahme in den Cache, wird das Paket weitergesendet und durch *Kontrolle* (siehe dort) überprüft, ob sich der Status dadurch ändert.

Der Initiator P_I sendet neben der eigentlichen Modus-Anforderung auch seinen aktiven Modus P_C und eine globale Sequenznummer P_{SG}. Mit diesen Informationen wird überprüft, ob beim Initiator derjenige Modus aktiv ist, der aus Sicht des empfangenden Knotens der gültige Netz-Modus ist. Ist der aktive Modus P_C kleiner als der Status-Modus S_M des empfangenden Knotens (d. h. er hat eine kleinere Kennung und ist „weniger prior“) und ist die globale Sequenznummer P_{SG} größer als S_{SG} (das Paket ist „aktuell“), wird der Status des empfangenden Knotens von diesem gesendet (mit allen Informationen des verantwortlichen Cache-Eintrags und mit entsprechend um eins erhöhter globalen Sequenznummer).

Beim Auslaufen einer Gültigkeitsdauer (Timeout) gelten ähnliche Regeln wie beim Modus-Wechsel-Protokoll *Zeroknowledge*. Auch hier führt das Auslaufen der Gültigkeitsdauer der internen Modus-Anforderung zu keinen weiteren Aktionen, da diese ggf. schon im Status repräsentiert ist. Gleiches gilt für das Auslaufen der Gültigkeitsdauer eines Cache-Eintrags. Läuft die Gültigkeitsdauer des Status aus, wird wie bei *Zeroknowledge* zu *Kontrolle* gewechselt.

Die *Kontrolle* übernimmt die Festlegung des aktuellen Status und überprüft, ob bei gegebenem Status andere Knoten über eine eventuell vorhandene interne Modus-Anforderung informiert werden müssen. Der aktuelle Status eines Knotens ergibt sich unmittelbar aus seinem Cache. Derjenige Cache-Eintrag mit dem größten Modus verantwortet den aktuellen Status, bestimmt somit mit seinen Einträgen C_I, C_M, C_V die Statusinformationen Initiator S_I, Modus S_M und Dauer S_V. Gleich wie beim Modus-Wechsel-Protokoll *Zeroknowledge* wird genau dann ein neues Modus-Anforderungspaket gesendet, wenn die interne Modus-Anforderung D_M größer ist als der aktive Status-Modus S_M und wenn eine gewisse Mindestdauer $D_{V_{min}}$ vorliegt.

4.8 Beweis der Konvergenz, Konsistenz und Korrektheit

Im Folgenden werden für die zuvor vorgestellten Modus-Wechsel-Protokolle *Zeroknowledge* und *Multiknowledge* die Konvergenz, Konsistenz und Korrektheit bewiesen. Bei der Konvergenz soll nachgewiesen werden, dass die Protokolle stets einen Zustand herstellen, bei dem sich der aktive Modus jedes Knotens nicht mehr ändert. Der Konsistenzbeweis soll nachweisen, dass in diesem konvergierten Zustand jeder Knoten den gleichen aktiven Modus hat. Der Korrektheitsbeweis soll schließlich nachweisen, dass der gleiche, aktive Modus jedes Knotens dem gültigen Netz-Modus entspricht. Von folgenden Voraussetzungen wird ausgegangen:

- Nachrichten (Modus-Anforderungspakete) gehen nicht verloren.
- Neue interne Modus-Anforderungen werden nur zu Beginn des betrachteten Zeitraums angefordert.
- In der Zwischenzeit gibt es keine neuen internen Modus-Anforderungen.
  (Anmerkung: Neue Modus-Anforderungspakete werden gesendet, soweit es die Regeln des Protokolls vorschreiben. Somit bezieht der Beweis das Senden neuer Modus-Anforderungspakete mit ein, was auch *neue* interne Modus-Anforderungen miteinschließen würde. Die Beweisführung gilt deswegen auch für Situationen mit *neuen* internen Modus-Anforderungen. Zur besseren Übersichtlichkeit wurde aber darauf verzichtet.)
- Die Geltungsdauer aller internen Modus-Anforderungen sei für den betrachteten Zeitraum ausreichend (d. h. größer als V_{min}).

4.8.1 Beweis für *Zeroknowledge*

Beweisskizze

Zu einem gewissen Zeitpunkt sei das Netz konsistent. D. h. jeder Knoten hat einen aktuellen Modus, es gibt keine Modus-Anforderungspakete, und es gibt keine internen Modus-Anforderungen, die das Senden neuer Modus-Anforderungspakete auslösen würden. Zu diesem Zeitpunkt werden somit keinerlei Pakete versendet und jeder Knoten hat den gleichen, aktuell geltenden (Netz-)Modus.

Zu einem Folgezeitpunkt haben ein oder mehrere Knoten eine neue Modus-Anforderung. Die Modus-Anforderungen sind beliebig in Anzahl und Inhalt (d. h. verschiedene Knoten können verschiedene Modi anfordern).

Jeder Knoten, der eine höhere Modus-Anforderung hat als der aktuell geltende Netz-Modus, sendet ein Modus-Anforderungspaket mit einer Sequenznummer. D. h. es gibt eine endliche Menge von „initialen“ Modus-Anforderungspaketen.

Jedes Modus-Anforderungspaket wird von jedem Knoten maximal einmal weitergesendet. Weitere Modus-Anforderungspakete als Antwort auf „initiale“ Modus-Anforderungspakete werden von Knoten gesendet, die ein Modus-Anforderungspaket erhalten, das einen kleineren Modus enthält als die eigene, interne Modus-Anforderung. Es wird nachgewiesen, dass es nur eine endliche Anzahl von „Antworten“ gibt, und damit die gesamte Zahl endlich ist.

Jeder Modus hat eine Kennung. Da jeder Knoten nur Modus-Anforderungspakete annimmt, deren Modus-Kennung größer ist, als sein aktueller Modus, nimmt die Kennung seines aktuellen Modus monoton zu. Es wird nachgewiesen, dass das Modus-Anforderungspaket mit der höchsten Modus-Kennung dem Netz-Modus entspricht.

Definitionen

Folgende Definitionen sollen gelten:

1. $\mathcal{D}$: Menge aller internen Modus-Anforderungen.
2. $D(x)$: Modus-Anforderung des Knotens x.
3. $D(x)_M$: Modus der Modus-Anforderung des Knotens x.

Korollar 4.1

Jedes Modus-Anforderungspaket wird maximal einmal von jedem Knoten weitergesendet.

Beweis: Ein Modus-Anforderungspaket P, das der zu betrachtende Knoten erhält, habe den Inhalt $P = (P_M, P_V, P_S)$. Der betrachtete Knoten habe den Status $S = (S_M, S_V, S_S)$. Folgende Fälle können dann unterschieden werden:

1. $P_S < S_S$:
   Nach der Analyseregel (1) wird dieses Paket nicht weitergesendet.
2. $P_S = S_S \wedge (P_M \leq S_M)$:
   Nach der Analyseregel (2.1) wird dieses Paket nicht weitergesendet.
3. $P_S = S_S \wedge (P_M > S_M)$:
   Nach der Analyseregel (2.2) wird der Status auf $S'_M := P_M, S'_V := P_V$ geändert ($S'_S = S_S = P_S$ bleibt gleich). Wird dieses Modus-Anforderungspaket ein weiteres Mal empfangen, gilt Fall 2 und das Paket wird nicht weitergesendet.
4. $P_S > S_S$:
   Nach der Analyseregel (3) wird der Status auf $S'_M := P_M, S'_V := P_V, S'_S := P_S$ geändert. Folgende Fälle können dann auftreten:
   a) Der Status ist kleiner als die interne Modus-Anforderung: Nach den Protokollregeln wird ein neues Modus-Anforderungspaket mit $P' := (..., P'_S := S'_S+1)$ gesendet und der Status entsprechend angepasst ($S''_S := P'_S = S'_S+1$). Wird Paket P ein weiteres Mal empfangen, gilt Fall 1 und das Paket P wird nicht weitergesendet.
   b) Ansonsten: Das Paket P wird weitergesendet. Wird das Paket ein weiteres Mal empfangen, gilt Fall 2 ($P_S = S'_S \wedge P_M = S'_M$) und das Paket wird nicht weitergesendet.

□

Korollar 4.2

Der Knoten mit der kleinsten internen Modus-Anforderung sendet maximal ein Modus-Anforderungspaket.

Beweis: Nach den Protokollregeln sendet ein Knoten ein Modus-Anforderungspaket P, wenn seine interne Modus-Anforderung größer ist als sein Status. Folgende Fälle können unterschieden werden:

1. Es gibt keine höhere interne Modus-Anforderung: Kein anderer Knoten, der das Modus-Anforderungspaket erhält, sendet seine eigene interne Modus-Anforderung. Der betrachtete Knoten erhält somit keine fremde Modus-Anforderung. Somit sendet er kein Modus-Anforderungspaket mehr.
2. Es gibt einen Knoten mit einer höheren internen Modus-Anforderung: Alle Knoten mit höherer interner Modus-Anforderung, die das Modus-Anforderungspaket

P erhalten, senden ein neues Modus-Anforderungspaket P' mit höherem Modus und höherer Sequenznummer ($P'_S := P_S + 1$). Erhält der betrachtete Knoten eines dieser neuen Modus-Anforderungspakete, setzt er seinen Status auf den höheren Status. Da nach den Voraussetzungen keine neuen internen Modus-Anforderungen im Beweiszeitraum angefordert werden und die Gültigkeitsdauer V_{min} im Beweiszeitraum ausreicht, kann der Status-Modus nicht mehr kleiner werden. Nach den Protokollregeln sendet der betrachtete Knoten somit kein weiteres eigenes Modus-Anforderungspaket.

□

4.8.2 Konvergenz von *Zeroknowledge*

Jeder Knoten x, der eine interne Modus-Anforderung $D(x) \in \mathcal{D}$ hat, und dessen interne Modus-Anforderung größer als sein aktiver Modus ist, sendet ein Modus-Anforderungspaket. Die Anzahl n_0 dieser initialen Modus-Anforderungspakete der „Runde 0“ kann deswegen durch

$$n_0 \leq |\mathcal{D}|$$

abgeschätzt werden.

Jeder Knoten x, der ein Modus-Anforderungspaket P erhält, dessen angeforderter Modus P_M kleiner ist als die interne Modus-Anforderung $D(x)_M$, sendet eine neues Modus-Anforderungspaket. Mindestens der Knoten mit der kleinsten internen Modus-Anforderung schickt nach Korollar 4.1 kein weiteres Modus-Anforderungspaket. Die Anzahl n_1 dieser (neuen) Modus-Anforderungspakete der Runde 1 kann wie folgt abgeschätzt werden (jeder Knoten, außer derjenige mit der kleinsten Modus-Anforderung, sendet ein neues Paket):

$$n_1 \leq n_0(n_0 - 1)$$

Als Antwort auf diese neuen Modus-Anforderungspakete senden wiederum alle Knoten, die ein Modus-Anforderungspaket mit kleinerem Modus erhalten, ein neues Modus-Anforderungspaket. Da der Knoten mit der bisher kleinsten Modus-Anforderung kein Paket gesendet hat, entfällt dieser in der Betrachtung. Demnach schickt der Knoten mit der jetzt kleinsten internen Modus-Anforderung kein weiteres Modus-Anforderungspaket. Die Anzahl n_2 der Modus-Anforderungspakete dieser Runde 2 kann deswegen mit

$$n_2 \leq n_1(n_1 - 1)$$

abgeschätzt werden.

Für jede folgende Runde i gilt auf analoge Weise, dass

$$n_i \leq n_{i-1}(n_{i-1} - 1)$$

Modus-Anforderungspakete gesendet werden.

Da in jeder Runde die Anzahl der betrachteten internen Modus-Anforderungen streng monoton fällt (jeweils der Knoten mit der kleinsten internen Modus-Anforderung sendet kein Paket mehr), ist die Anzahl der Runden auf $|\mathcal{D}|$ begrenzt. Somit gilt für die Gesamtzahl der Modus-Anforderungspakete:

$$n = n_0 + n_1 + \ldots + n_{|\mathcal{D}|}$$

Da die Anzahl n der gesendeten Modus-Anforderungen nach oben beschränkt ist, und nach Korollar 4.2 jeder Knoten jedes Modus-Anforderungspaket maximal einmal weitersendet, werden spätestens nach Abschluss aller Runden keine Pakete mehr gesendet.

Jeder Knoten hat somit den Modus des Modus-Anforderungspakets mit dem höchsten Modus, das er empfangen hat, als aktiven Modus. Da keine Pakete mehr gesendet werden, ändert sich der aktive Modus nicht mehr und jeder Knoten hat einen festen Zustand (die Gültigkeitsdauer V_{min} ist laut Voraussetzung genügend groß). Das Protokoll terminiert. Das Netz ist konvergiert. □

4.8.3 Konsistenz von *Zeroknowledge*

Für den folgenden Beweis gelten die gleichen Voraussetzung wie beim Beweis für die Konvergenz von *Zeroknowledge* (siehe Abschnitt 4.8.2). Zusätzlich wird von folgenden Voraussetzungen ausgegangen:

1. Jeder Knoten hat zu Beginn die gleiche Sequenznummer. Diese wird mit N_{0S} bezeichnet. Somit gilt der Status der Sequenznummer $S(x)_S = N_{0S}$ für jeden Knoten x.
2. Jeder Knoten hat zu Beginn den gleichen aktiven Modus. Dieser wird mit N_{0M} bezeichnet. Somit gilt der Status des Modus $S(x)_M = N_{0M}$ für jeden Knoten x.

Gibt es keine interne Modus-Anforderung $D(x) \in \mathcal{D}$, die größer ist als der aktive Modus N_{0M}, sendet kein Knoten ein Modus-Anforderungspaket und das Protokoll

terminiert. Jeder Knoten hat dann den gleichen aktiven Modus N_{0M} und die gleiche Sequenznummer N_{0S}. Das Netz ist konsistent.

Andernfalls sendet in der Runde 0 jeder Knoten x, der eine interne Modus-Anforderung $D(x) \in \mathcal{D}$ hat, und dessen interne Modus-Anforderung größer als sein aktiver Modus ist, ein Modus-Anforderungspaket $P(x)_0$ mit der Sequenznummer $P(x)_{0S} = N_{1S} := N_{0S} + 1$. Dabei gibt es mindestens ein Modus-Anforderungspaket mit dem Modus P_{0maxM} und Sequenznummer P_{0S}, für das gilt: $P_{0maxM} \geq P(x)_{0M} \forall x$.

Jeder Knoten x, der ein Modus-Anforderungspaket der Runde 0 erhält, das einen kleineren Modus enthält als die interne Modus-Anforderung von x, sendet in Runde 1 ein Modus-Anforderungspaket $P(x)_1$ mit der Sequenznummer $P(x)_{1S} = N_{2S} := N_{1S}+1$. Auch hier gilt, dass mindestens ein Modus-Anforderungspaket existiert, für das gilt: $P_{1maxM} \geq P(x)_{1M} \forall x$.

Generell gilt für jede Runde i, die stattfindet (d. h. das Protokoll noch nicht vorher terminiert hat):

- Jedes Modus-Anforderungspaket $P(x)_i$ hat die Sequenznummer $P(x)_{iS} = N_{(i+1)S} := N_{iS} + 1$.
- Die Sequenznummer $P(x)_{iS}$ ist größer als die aller vorherigen Runden.
- Es gibt mindestens ein Modus-Anforderungspaket, für das gilt: $P_{imaxM} \geq P(x)_{iM} \forall x$.

Da die Anzahl der Runden begrenzt ist (siehe Beweis der Konvergenz von *Zeroknowledge* in Abschnitt 4.8.2), gibt es eine letzte Runde n. Es gibt ein Modus-Anforderungspaket $P(x)_n = (P(x)_{nS}, P(x)_{nM})$, für das gilt $P(x)_{nM} = P_{imaxM} : P_{imaxM} \geq P(x)_{nM} \forall x$. Da dieses Paket die höchste Sequenznummer und den höchsten Modus (mit dieser Sequenznummer) enthält, erhält jeder Knoten dieses Paket mindestens einmal und leitet es genau einmal weiter. Somit hat jeder Knoten x nach dieser letzten Runde den gleichen aktiven Modus $S(x)_M = P(x)_{nM}$ und die gleiche Sequenznummer $S(x)_S = P(x)_{nS}$. Das Netz ist konsistent. □

4.8.4 Korrektheit von *Zeroknowledge*

Für den folgenden Beweis gelten die gleichen Voraussetzungen wie im Beweis für die Konvergenz und die Konsistenz von *Zeroknowledge* (siehe Abschnitt 4.8.2 und Abschnitt 4.8.3).

Ein Netz ist dann korrekt, wenn jeder Knoten x den Netz-Modus N_M als aktiven Modus hat. Zur Erinnerung sei hier die formale Definition des Netz-Modus wiederholt:

$$N_M = \begin{cases} D(x)_M | D(x)_M \geq D(y)_M \forall D(x)_M, D(y)_M \in \mathcal{D} & \text{, falls } \mathcal{D} \neq \emptyset, \\ M_0 & \text{, sonst.} \end{cases}$$

wobei $D(x)_M, D(y)_M$ = Modusanforderung eines Knotens x beziehungsweise y
$\mathcal{D}$ = Menge aller Modus-Anforderungen
M_0 = Standard-Modus

Behauptung: Nach der letzten Runde n (siehe Beweis zur Konsistenz von *Zeroknowledge* in Abschnitt 4.8.3) ist der konsistente Modus auch der Netz-Modus.

Beweis durch Widerspruch:

Annahme: Es gibt in der letzten Runde n einen Knoten x mit einer höheren internen Modus-Anforderung $D(x)$ als das Modus-Anforderungspaket P_n mit $P_n = (P_{nmaxM}, P_{nS})$.

Da jeder Knoten dieses erhält (siehe Beweis zur Konsistenz von *Zeroknowledge* in Abschnitt 4.8.3), erhält dies auch Knoten x. Dieser sendet nach den Protokollregeln ein Modus-Anforderungspaket $P'(x)$ mit $P'(x) = (P'(x)_M := D(x)_M, P'(x)_S := P_{nS} + 1)$. Dies entspricht der Runde $n + 1$. Runde n ist somit nicht die letzte. Widerspruch zur Annahme. □

4.8.5 Beweis für *Multiknowledge*

Beweisidee

Zu einem gewissen Zeitpunkt sei das Netz konsistent und konvergiert. D. h. es gibt einen gültigen und konsistenten aktiven Netz-Modus. Zu einem Folgezeitpunkt gebe es eine definierte Menge aller internen Modus-Anforderungen. Jeder Knoten sendet seine Modus-Anforderung. Jede Modus-Anforderung, die für den aktiven Modus relevant sein könnte (und damit in den Cache aufgenommen wird), erreicht durch die Modusregeln jeden Knoten. Der aktive Modus muss damit für jeden Knoten gleich sein.

Definitionen

Folgende Definitionen sollen gelten:

1. $\mathcal{D}$: Menge aller internen Modus-Anforderungen.
2. $D(x)$: Modus-Anforderung des Knotens x.
3. $D(x)_M$: Modus der Modus-Anforderung des Knotens x.

Korollar 4.3

Jedes Modus-Anforderungspaket wird maximal einmal von jedem Knoten weitergesendet.

Beweis: Ein Modus-Anforderungspaket P, das der zu betrachtende Knoten erhält, habe den Inhalt $P = (P_I, P_M, P_V, P_{SL})$. Folgende Fälle können dann unterschieden werden:

1. Ein Paket von einem bekannten Initiator, aber mit kleinerer oder gleicher lokaler Sequenznummer wird empfangen:
   $\exists x : P_I = C(x)_I \wedge P_{SL} \leq C(x)_S$:
   Aufgrund Analyseregel (3) wird dieses Paket verworfen.
2. Ein Paket von einem bekannten Initiator und größerer lokaler Sequenznummer wird empfangen:
   $\exists x : P_I = C(x)_I \wedge P_{SL} > C(x)_S$:
   Aufgrund Analyseregel (1) wird das Paket weitergesendet. Wird das Paket ein weiteres Mal empfangen, gilt Fall 1 und das Paket wird dann verworfen.
3. Ein Paket von einem unbekannten Initiator wird empfangen und ein Cache-Platz ist frei:
   $(\forall x : P_I \neq C(x)_I) \wedge (\exists y : C(y) = \oslash)$:
   Aufgrund Analyseregel (2.1) wird das Paket weitergesendet. Wird das Paket ein weiteres Mal empfangen, gilt Fall 1 und das Paket wird dann verworfen.
4. Ein Paket von einem unbekannten Initiator wird empfangen und es gibt keinen Cache-Platz mit kleinerem Modus:
   $(\forall x : P_I \neq C(x)_I) \wedge (\forall y : C(y)_M > P_M)$:
   Aufgrund Analyseregel (3) wird das Paket verworfen.
5. Ein Paket von einem unbekannten Initiator wird empfangen und es gibt einen Cache-Platz mit kleinerem Modus:
   $(\forall x : P_I \neq C(x)_I) \wedge (\forall y : C(y)_M < P_M)$:
   Aufgrund Analyseregel (2.2) wird das Paket weitergesendet. Wird das Paket ein weiteres Mal empfangen, gilt Fall 1 und das Paket wird dann verworfen.

6. Ein Paket von einem unbekannten Initiator wird empfangen und es gibt einen Cache-Platz mit gleichem Modus, aber mit längerer Gültigkeitsdauer:
   $(\forall x : P_I \neq C(x)_I) \wedge (\forall y : (C(y)_M > P_M) \vee (C(y)_M = P_M \wedge C(y)_V \geq P_V))$:
   Aufgrund Analyseregel (3) wird das Paket verworfen.
7. Ein Paket von einem unbekannten Initiator wird empfangen und es gibt einen Cache-Platz mit gleichem Modus mit kürzerer Dauer:
   $(\forall x : P_I \neq C(x)_I) \wedge (\forall y : (C(y)_M > P_M) \vee (C(y)_M = P_M \wedge C(y)_V < P_V))$:
   Aufgrund Analyseregel (2.3) wird das Paket weitergesendet. Wird das Paket ein weiteres Mal empfangen, gilt Fall 1 und das Paket wird dann verworfen.

□

Korollar 4.4

Jeder Knoten sendet maximal ein Modus-Anforderungspaket.

Beweis: Nach den Protokollregeln sendet ein Knoten ein Modus-Anforderungspaket P, wenn seine interne Modus-Anforderung größer ist als sein Status. Nach den Protokollregeln sendet seine Modus-Anforderung kein zweites Mal. (Anmerkung: Dies gilt insbesondere deswegen, da kein Knoten seine Modus-Anforderung verringern darf.) □

4.8.6 Konvergenz für *Multiknowledge*

Jeder Knoten x, der eine interne Modus-Anforderung $D(x) \in \mathcal{D}$ hat, und dessen interne Modus-Anforderung größer als sein aktiver Modus ist, sendet ein Modus-Anforderungspaket. Die Anzahl n_0 dieser initialen Modus-Anforderungspakete der „Runde 0“ kann deswegen durch

$$n_0 \leq |\mathcal{D}|$$

abgeschätzt werden.

Nach Korollar 4.3 sendet jeder Knoten jede Modus-Anforderung maximal einmal weiter. Nach Korollar 4.4 sendet kein Knoten eine weitere Modus-Anforderung. Damit ist die Gesamtzahl der gesendeten Modus-Anforderungen nach oben beschränkt. Das Protokoll terminiert und konvergiert. □

4.8.7 Konsistenz von *Multiknowledge*

Für den folgenden Beweis gelten die gleichen Voraussetzungen wie beim Beweis für die Konvergenz von *Multiknowledge* (siehe Abschnitt 4.8.6). Zusätzlich wird von folgender Voraussetzungen ausgegangen:

Alle Knoten haben zu Beginn einen äquivalenten Cache.

Definition: Äquivalenz zweier Caches

Zwei Cache-Plätze $C(x)$ und $C(x')$ sind äquivalent, wenn

1. $C(x)_M = C(x')_M$ und
2. $C(x)_V = C(x')_V$.

Zwei Caches C und C' sind äquivalent, wenn

1. zu jedem Cache-Platz $C(x)$ ein äquivalenter Cache-Platz $C'(y)$ existiert, und
2. zu jedem Cache-Platz $C'(x)$ ein äquivalenter Cache-Platz $C(y)$ existiert.

Lemma 4.1

Wenn zwei Modus-Anforderungspakete P und P' von einem Knoten empfangen werden, ist sein resultierender Cache bei der Ankunftsreihenfolge $P < P'$ äquivalent zum resultierenden Cache bei der Ankunftsreihenfolge $P' < P$.

Beweis:
Es folgt eine Fallunterscheidung aufgrund der zutreffenden Protokollregeln. Dabei wird unterstellt, dass die bei der Fallunterscheidung zutreffende Regel für den unveränderten Cache gilt (eventuell ändert somit die Ankunft des ersten Modus-Anforderungspakets die zutreffende Regel des zweiten Pakets). Folgende Fälle werden unterschieden:

1. Beide Modus-Anforderungspakete P und P' können im Cache aufgenommen werden wegen Protokollregeln (1) bis (2.3).
   a) Auf P und P' trifft Regel (1) zu.
   b) Auf P trifft Regel (1) zu, auf P' Regel (2.1).
   c) Auf P trifft Regel (1) zu, auf P' Regel (2.2).

   d) Auf P trifft Regel (1) zu, auf P' Regel (2.3).
   e) Auf P trifft Regel (2.1) zu, auf P' Regel (2.1).
   f) Auf P trifft Regel (2.1) zu, auf P' Regel (2.2).
   g) Auf P trifft Regel (2.1) zu, auf P' Regel (2.3).
   h) Auf P trifft Regel (2.2) zu, auf P' Regel (2.2).
   i) Auf P trifft Regel (2.2) zu, auf P' Regel (2.3).
   j) Auf P trifft Regel (2.3) zu, auf P' Regel (2.3).

2. Das Modus-Anforderungspaket P kann im Cache aufgenommen werden wegen Protokollregeln (1) bis (2.3); das Modus-Anforderungspaket P' nicht wegen Protokollregel (3).
   a) Auf P trifft Regel (1) zu.
   b) Auf P trifft Regel (2.1) zu.
   c) Auf P trifft Regel (2.2) zu.
   d) Auf P trifft Regel (2.3) zu.
3. Beide Modus-Anforderungspakete P und P' können nicht im Cache aufgenommen werden wegen Protokollregel (3).

zu Fall 1a:
Dieser Fall kann nicht eintreten, da in diesem Fall $P_I = P'_I$ gelten müsste und dies nach den Beweisvoraussetzungen nicht eintritt (ein Knoten kann innerhalb der betrachteten Zeit nicht unterschiedliche Modus-Anforderungen stellen).

zu Fall 1b:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: der entsprechende Cache-Platz x mit $C(x)_I = P_I$ wird aktualisiert.
2. Auswirkung von P': ein Cache-Platz y ($\neq x$) mit $C(y) = \oslash$ wird belegt.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 1c:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: der entsprechende Cache-Platz x mit $C(x)_I = P_I$ wird aktualisiert.
2. Auswirkung von P': ein Cache-Platz y ($\neq x$) mit $C(y)_M < P'_M$ wird belegt.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 1d:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: der entsprechende Cache-Platz x mit $C(x)_I = P_I$ wird aktualisiert.
2. Auswirkung von P': ein Cache-Platz y ($\neq x$) mit $C(y)_M = P'_M \wedge C(y)_V < P'_V$ wird belegt.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 1e:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: ein Cache-Platz x mit $C(x) = \oslash$ wird belegt.
2. Auswirkung von P': ein Cache-Platz y ($\neq x$) mit $C(y) = \oslash$ wird belegt.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 1f:
Es gilt zwingenderweise die Ankunftsreihenfolge P nach P', da für P noch ein Cache-Platz frei ist, für P' aber nicht. Deswegen existiert für diesen Fall keine Wahlmöglichkeit der Ankunftsreihenfolge. Die resultierenden Caches sind deswegen äquivalent.

zu Fall 1g:
Es gilt die gleiche Überlegung wie bei Fall 1f.

zu Fall 1h:
Es gelte o.b.d.A. $P_M < P'_M$. Für die Ankunftsreihenfolge P vor P' ergibt sich:

1. Auswirkung von P: der Cache-Platz x mit dem kleinsten Modus $C(x)_M$ wird aktualisiert.
2. Auswirkung von P': der Cache-Platz x (sic!) wird erneut aktualisiert.

Für die Ankunftsreihenfolge P' vor P ergibt sich:

1. Auswirkung von P': der Cache-Platz x mit dem kleinsten Modus $C(x)_M$ wird aktualisiert.
2. Auswirkung von P: da $P_M < P'_M$ gilt, ist kein Cache-Platz mehr frei. Das Paket wird verworfen.

Der Cache-Platz x ist somit in beiden Fällen mit der gleichen Modus-Anforderung belegt. Die resultierenden Caches sind deswegen äquivalent.

zu Fall 1i:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: der Cache-Platz x mit dem kleinsten Modus $C(x)_M$ wird aktualisiert.
2. Auswirkung von P': der Cache-Platz y ($\neq x$) mit $C(y)_M = P_M \wedge C(y)_V < P_V$ wird belegt.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 1j:
Es gelte o.b.d.A $P_M = P'_M \wedge P_V < P'_V$. Für die Ankunftsreihenfolge P vor P' ergibt sich:

1. Auswirkung von P: der Cache-Platz x mit dem Modus $C(x)_M = P_M$ und der kleinsten Gültigkeitsdauer $C(x)_V$ wird aktualisiert.
2. Auswirkung von P': der Cache-Platz x (sic!) wird erneut aktualisiert.

Für die Ankunftsreihenfolge P' vor P ergibt sich:

1. Auswirkung von P': der Cache-Platz x mit dem Modus $C(x)_M = P_M$ und der kleinsten Gültigkeitsdauer $C(x)_V$ wird aktualisiert.
2. Auswirkung von P: da $P_V < P'_V$ gilt, ist kein Cache-Platz mehr frei. Das Paket wird verworfen.

Der Cache-Platz x ist somit in beiden Fällen mit der gleichen Modus-Anforderung belegt. Die resultierenden Caches sind deswegen äquivalent.

zu Fall 2a:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: der entsprechende Cache-Platz x mit $C(x)_I = P_I$ wird aktualisiert.
2. Auswirkung von P': keine Aktualisierung oder Belegung eines Cache-Platzes, da Regel (3) zutrifft.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 2b:
Es gilt zwingenderweise die Reihenfolge P vor P', da Paket P aufgrund eines freien Cache-Platzes aufgenommen wird, Paket P' jedoch nicht. Die jeweils resultierenden Caches sind äquivalent.

zu Fall 2c:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P: der Cache-Platz x mit dem kleinsten Modus $C(x)_M$ wird aktualisiert.
2. Auswirkung von P': keine Aktualisierung oder Belegung eines Cache-Platzes, da Regel (3) zutrifft.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 2d:
Unabhängig von der Ankunftsreihenfolge gilt:

1. Auswirkung von P': der Cache-Platz x mit dem Modus $C(x)_M = P_M$ und der kleinsten Gültigkeitsdauer $C(x)_V$ wird aktualisiert.
2. Auswirkung von P': keine Aktualisierung oder Belegung eines Cache-Platzes, da Regel (3) zutrifft.

Die jeweils resultierenden Caches sind äquivalent.

zu Fall 3:
Beide Pakete werden wegen Regel (3) nicht in den Cache aufgenommen. Die resultierenden Caches sind äquivalent. □

Beweis der Konsistenz mit Lemma 4.1

Paarweises Tauschen der Ankunftsreihenfolge ergibt nach Lemma 4.1 äquivalente Caches. Die Ankunftsreihenfolgen von Modus-Anforderungspaketen können durch paarweises Tauschen ineinander überführt werden. Die jeweils resultierenden Caches sind deshalb äquivalent. Aus der Äquivalenz aller Caches folgt die Konsistenz. □

4.8.8 Korrektheit von *Multiknowledge*

Für den folgenden Beweis gelten die gleichen Voraussetzungen wie im Beweis für die Konvergenz und die Konsistenz von *Multiknowledge* (siehe Abschnitt 4.8.6 und Abschnitt 4.8.7).

Ein Netz ist dann korrekt, wenn jeder Knoten x den Netz-Modus N_M als aktiven Modus hat. Zur Erinnerung sei hier die formale Definition des Netz-Modus wiederholt:

$$N_M = \begin{cases} D(x)_M | D(x)_M \geq D(y)_M \forall D(x)_M, D(y)_M \in \mathcal{D} & \text{, falls } \mathcal{D} \neq \emptyset, \\ M_0 & \text{, sonst.} \end{cases}$$

wobei $D(x)_M, D(y)_M$ = Modusanforderung eines Knotens x beziehungsweise y
$\mathcal{D}$ = Menge aller Modus-Anforderungen
M_0 = Standard-Modus

Behauptung: Der konsistente Modus ist auch der Netz-Modus.

Beweis durch Widerspruch:

Annahme: Es gibt einen Knoten x mit einer höheren internen Modus-Anforderung $D(x)_M$ als der höchste Cache-Platz $C(y)_M$.

Der Knoten x hat nach den Modusregeln ein Modus-Anforderungspaket mit der Modus-Anforderung $D(x)$ gesendet. Da jeder Knoten dieses erhält (siehe Beweis zur Konsistenz von *Multiknowledge* in Abschnitt 4.8.7), ist diese auch im konsistenten Cache enthalten. Widerspruch zur Annahme. □

4.9 Evaluation

Im Folgenden sollen die oben entwickelten Protokolle zum Modus-Wechsel simulativ untersucht und evaluiert werden. Zuerst wird das Verhalten der Modus-Wechsel-Protokolle ohne Einfluss von anderen Anwendungen untersucht. Zum einen wird das Verhalten im zeitlichen Verlauf bei mehreren Modus-Wechseln gezeigt. Die wesentlichen Kennwerte sind dabei die Anzahl der verschickten Pakete, die nötig sind, um alle Knoten in den gewünschten Modus zu wechseln, und die Zeit, bis der gewünschte Modus erreicht wird. Zum anderen werden die genannten Kennzahlen der Modus-Wechsel-Protokolle bei variierender Cache-Größe und variierender Anzahl von zur Verfügung stehender Modi gegenübergestellt.

Anschließend wird die Effektivität eines Modus-Wechsels während der Aktivität anderer Anwendungen untersucht. Die Anwendung kommuniziert dabei entsprechend den oben vorgestellten, exemplarischen Modus-Implementierungen für energiesparende, schnelle, beziehungsweise robuste Kommunikation. Unter Ausnutzung der zur Verfügung stehenden Modus-Implementierungen soll der effektive und effiziente Modus-Wechsel nachgewiesen werden.

Die nachfolgenden Simulationen basieren auf Implementierungen der vorgestellten Protokolle im Netzwerksimulator ns2 [ns2]. In allen folgenden Simulationen werden 100 Knoten in einem Areal von 1000 m x 1000 m zufällig verteilt. Für MAC- und PHY-Schicht wird festgelegt, das die Kommunikationsreichweite der Knoten 150 m beträgt. Alle Knoten senden auf dem gleichen Kanal. Fehler treten auf dieser Schicht nur für den Fall auf, dass Knoten gleichzeitig oder überschneidend den Kanal belegen. Ausfälle von Knoten sind davon aber unbenommen. In Abschnitt 4.9.3 und 4.9.4 wird das Verhalten von Modus-Implementierungen und Modus-Wechsel-Protokollen bei Knotenausfall untersucht. Dabei fallen Knoten spontan aus und nehmen nicht mehr an MAC- oder Routing-Aufgaben teil.

Die Anzahl der Knoten wurde so groß gewählt, um für die Evaluierung der Konsistenz der Modus-Wechsel-Protokolle statistisch aussagekräftige Resultate zu erzielen, und genügend klein, um die Simulationen in sinnvoller Rechenzeit durchführen zu können. Feldgröße und Kommunikationsreichweite wurden so festgelegt, dass keine Partitionierungen auftreten und um „nicht-triviales“ Fluten zu erzielen (die Parameter ergeben ca. einen 10-Hop-Durchmesser in der Diagonalen).

Während die Modus-Wechsel-Protokolle ausschließlich die Broadcast-Fähigkeit der Knoten nutzen, basieren die Modus-Implementierungen auf der Standardimplemen-

tierung des Routing-Protokolls AODV (siehe [PBRD03]) des Netzwerksimulators ns2. AODV ist somit integraler Bestandteil der exemplarischen Modus-Implementierungen.

Die Simulationsparameter sind in Tab. 4.1 zusammengefasst. Für die nachfolgenden Simulationen sind alle wichtigen Simulationsparameter im Text erläutert. Eine komplette Auflistung der Parameter für jede Simulation ist in Anhang B zu finden.

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Zeroknowledge* und *Multiknowledge*
Kommunikationsprotokoll für Modus-Wechsel-Protokolle	Fluten (nach den Regeln des jeweiligen Protokolls)
Modus-Implementierungen	energiesparend, schnell, robust
Kommunikationsprotokoll für Modus-Implementierungen	AODV

Tabelle 4.1: Übersicht der Simulationsparameter; für eine komplette Auflistung aller Parameter jeder Simulation siehe auch Anhang B

4.9.1 Untersuchung der Modus-Wechsel-Protokolle im zeitlichen Verlauf

Im Folgenden werden die Modus-Wechsel-Protokolle *Zeroknowledge* und *Multiknowledge* mit den Cache-Größen 1 und 2 gegenübergestellt. Da das Modus-Wechsel-Protokoll *Zeroknowledge* keinen Cache besitzt, wird es abkürzend als Modus-Wechsel-Protokoll mit Cache-Größe 0 bezeichnet.

Zur Gegenüberstellung der drei Varianten werden, während einer Simulationszeit von 20 Sekunden, von zufällig ausgewählten Knoten Modus-Anforderungen initiiert, die jeweils 5 Sekunden gültig bleiben. Es werden drei zur Verfügung stehende Modi mit den Kennungen 0, 1 und 2 angenommen.

In Tab. 4.2 sind die gesammelten Modus-Anforderungen in der betrachteten Simulation aufgezählt. Die daraus resultierenden korrekten Modi sind in Abb. 4.8 dargestellt. Der korrekte Modus ermittelt sich dabei aus den gesamten zu einem gewissen Zeitpunkt existierenden Modus-Anforderungen und der Prioritätsordnung, die festlegt, dass

Zeit	Modus-Anforderung	Knoten
50,0	0	(kein)
50,8	1 (neu)	35
51,3	2 (neu)	43
53,7	2 (verlängert)	43
56,8	2 (neu)	3
58,7	0 (Time-0ut)	43
59,6	1 (neu)	16
61,8	0 (Time-0ut)	3
64,6	0 (Time-0ut)	16
67,6	1 (neu)	79
69,2	2 (neu)	25
74,2	0 (Time-0ut)	25

Tabelle 4.2: Modus-Anforderungen der Knoten während der betrachteten Einzelsimulation

Modus 1 eine höhere Priorität hat als Modus 0, und Modus 2 wiederum eine höhere Priorität hat als Modus 1. Ganz zu Beginn gilt ab der Simulationszeit 50 Sekunden der Modus 0. Bei Simulationszeit 51 Sekunden ergibt sich für kurze Zeit der Modus 1. Bei etwa 52 Sekunden wird dieser vom Modus 2 abgelöst, dessen Gültigkeit bei ca. 62 Sekunden abläuft. Die weiteren gültigen Modi können in Abb. 4.8 verfolgt werden.

Für die zuvor gegebenen Modus-Anforderungen ist das Verhalten des Modus-Wechsel-Protokolls *Zeroknowledge* (d. h. mit Cache-Größe 0) in Abb. 4.9 dargestellt. Zu Beginn sind definitionsgemäß alle Knoten im Standard-Modus 0. Nach einem kurzfristigen Wechsel in den korrekten Modus 1, wechseln alle Knoten ab 51,3 Sekunden

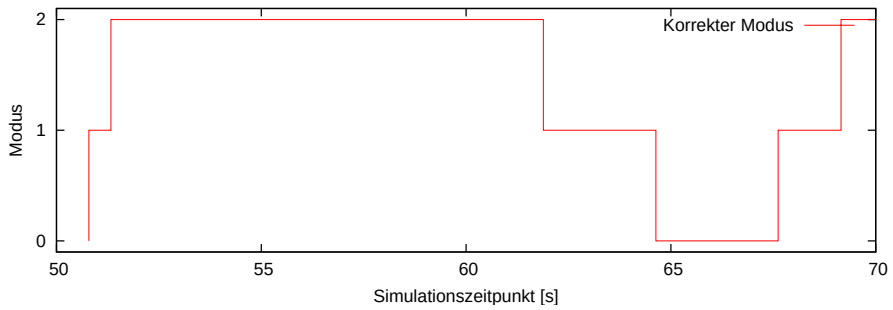

Abbildung 4.8: Korrekter Netzmodus während der betrachteten Einzelsimulation

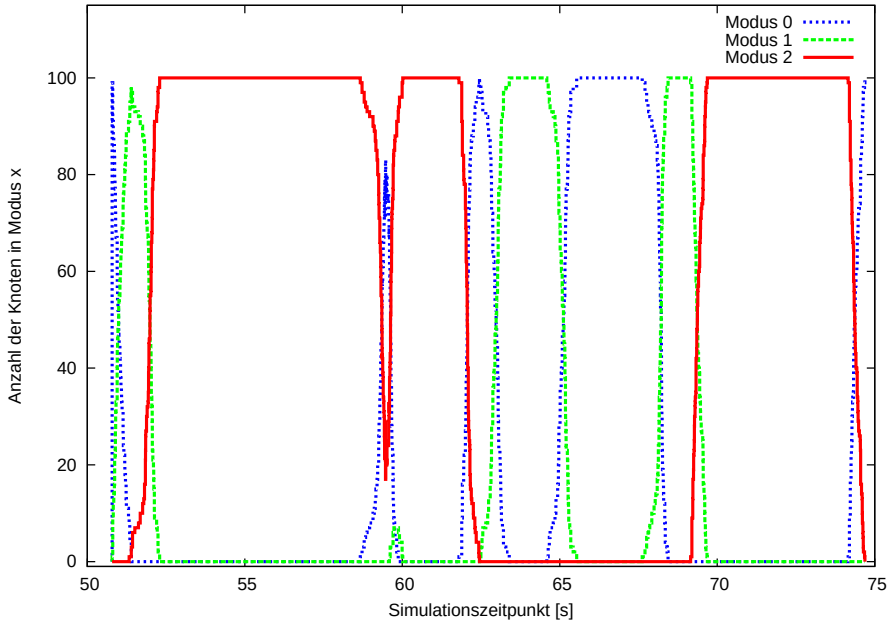

Abbildung 4.9: Modus der Knoten während der betrachteten Einzelsimulation mit Cachegröße 0; für die Simulationsparameter siehe auch Tab. B.1 im Anhang

aufgrund der Modus-Anforderung von Knoten 43 in den korrekten Modus 2. Während der Gültigkeit des Modus 2 hat ein weiterer Knoten dieselbe Modus-Anforderung, was folglich keine Auswirkung auf die anderen Knoten hat.

Zum Zeitpunkt 58,7 Sekunden läuft jedoch die Gültigkeitsdauer der Modus-Anforderung des Knotens 43 aus. Alle Knoten hatten die Modus-Anforderung von Knoten 43 empfangen und nehmen folglich wegen des Time-Outs den Standard-Modus 0 ein. Sobald derjenige Knoten, der immer noch eine Modus-Anforderung auf Modus 2 hat, ebenfalls durch den Time-Out auf Modus 0 gehen würde, verbreitet er die Modus-Anforderung Modus 2. Gleichzeitig sendet Knoten 16 eine Modus-Anforderung auf Modus 1. Nach kurzer Zeit und zwischenzeitlichem Wechsel einiger Knoten auf den (falschen) Modus 0 beziehungsweise 1 konvergiert das Netz auf den nach wie vor geltenden Modus 2.

In dieser Situation kann man das spezielle Verhalten des Modus-Wechsel-Protokolls bei Cache-Größe 0 erkennen. Da nur der Modus selbst bekannt ist, wechseln alle Knoten beim Auslaufen der Gültigkeit auf den Standard-Modus 0. Erst dann reagieren die Knoten mit einer anderen Modus-Anforderung. Dies hat zur Folge, dass sich zu diesen Zeitpunkten viele Knoten in einem falschen Modus befinden. Diese Situation wiederholt

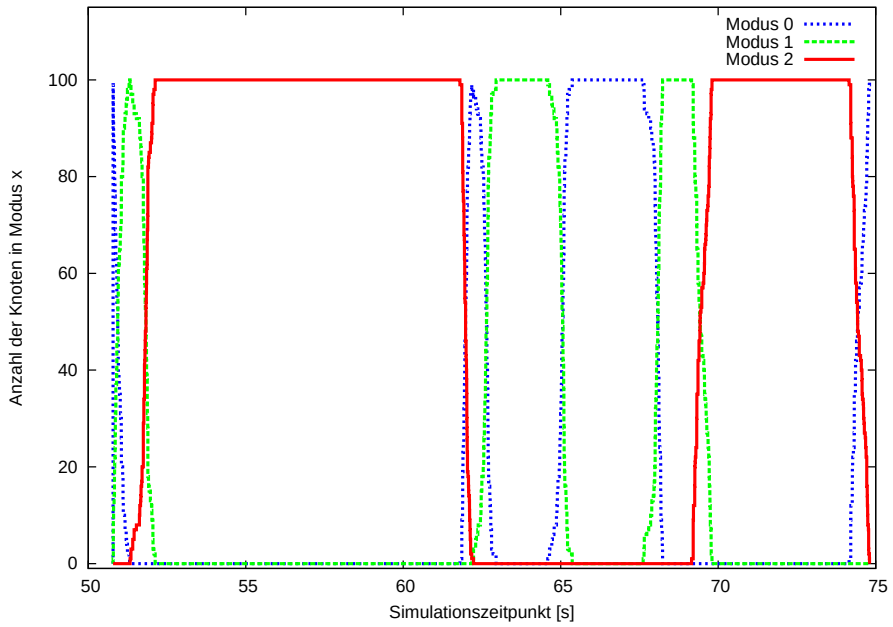

Abbildung 4.10: Modus der Knoten während der betrachteten Einzelsimulation mit Cachegröße 1; für die Simulationsparameter siehe auch Tab. B.2 im Anhang

sich zum Zeitpunkt 61,8 Sekunden, wo eigentlich Modus 1 gelten sollte, das Netz aber zuerst in den Standard-Modus 0 wechselt.

Das gleiche Experiment mit den gleichen Modus-Anforderungen ist in Abb. 4.10 abgebildet. Hier wurde aber das Modus-Wechsel-Protokoll *Multiknowledge* mit Cache-Größe 1 gewählt. Während der Beginn mit kurzzeitigen Wechseln in den korrekten Modus 1 und den darauffolgenden Wechseln in den Modus 2 gleich abläuft wie bei Cache-Größe 0, ist das Verhalten beim Auslaufen der Gültigkeit des Modus 2 bei Knoten 43 anders. Mit dem zur Verfügung stehen Cache-Platz wird die zweite Modus-Anforderung zum Zeitpunkt 56,8 Sekunden trotz dem schon bestehenden Modus 2 an alle Knoten mitgeteilt. Somit kennen die Knoten diese zweite Modus-Anforderung beim Auslaufen der Gültigkeit der ersten Modus-Anforderung zum Zeitpunkt 58,7 Sekunden. Alle Knoten bleiben deswegen im Gegensatz zur Cache-Größe 0 zu diesem Zeitpunkt konsistent im korrekten Modus 2. Nicht korrekt ist hingegen der zwischenzeitliche Wechsel auf den Modus 0 ab Zeitpunkt 61,8 Sekunden. Hier läuft die Modus-Anforderung des Knotens 3 ab. Die Knoten haben keine Information über die zwischenzeitlich erfolgte Modus-Anforderung nach Modus 1, da ihr Cache zum Zeitpunkt der Verbreitung bereits durch die höher-priore Modus-Anforderung nach Modus 2 von Knoten 3 belegt war. Sie wechseln

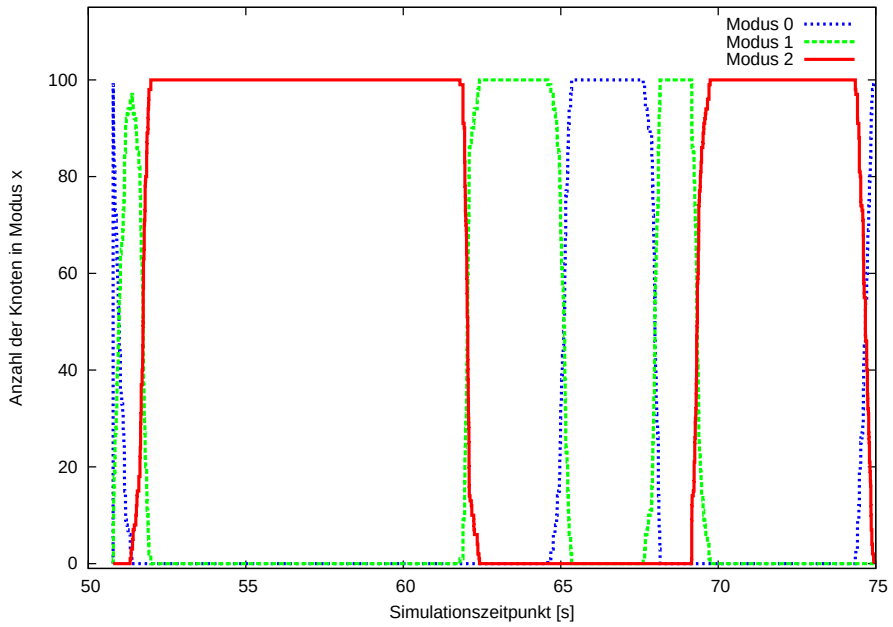

Abbildung 4.11: Modus der Knoten während der betrachteten Einzelsimulation mit Cachegröße 2; für die Simulationsparameter siehe auch Tab. B.3 im Anhang

deswegen vorübergehend in den Modus 0, bevor Knoten 16 seine Modus-Anforderung nach Modus 1 im Netz verbreitet.

Auch in Abb. 4.11 ist das gleiche Experiment mit den gleichen Modus-Anforderungen abgebildet, hier jedoch mit dem Modus-Wechsel-Protokoll *Multiknowledge* mit Cache-Größe 2. Auch hier läuft der Beginn identisch ab wie bei Cache-Größe 0 und 1. Zum Zeitpunkt 58,7 Sekunden wird ebenso wie bei Cache-Größe 1 ein zwischenzeitlicher Wechsel auf Modus 0 vermieden, da die Information über das Fortbestehen der Modus-Anforderung durch einen anderen Knoten im Cache vorgehalten wird. Durch den zusätzlichen zweiten Cache-Platz kann jedoch auch der zwischenzeitliche Wechsel auf Modus 0 zum Zeitpunkt 61,8 Sekunden vermieden werden. Stattdessen wechseln die Knoten direkt von Modus 2 auf den korrekten Modus 1. Dadurch finden in diesem Beispiel sämtliche Modus-Wechsel abzüglich der durch die Verbreitungszeit bedingten Latenz verzögerungsfrei statt.

Die Einzelbetrachtung des Verhaltens der drei verschiedenen Varianten des Modus-Wechsel-Protokolls lässt vermuten, dass bei sonst gleichen Randbedingungen der kor-

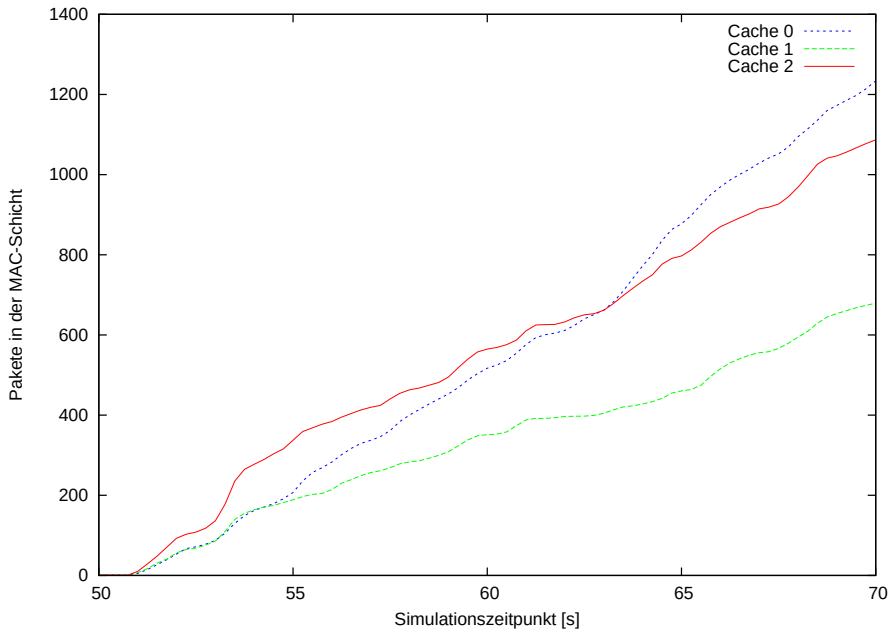

Abbildung 4.12: Kumulierte Anzahl der gesendeten MAC-Pakete (bei 10 Modus-Anforderungen während der Simulationszeit); für die Simulationsparameter siehe auch Tab. B.4 im Anhang

rekte Netzmodus mit zunehmender Cache-Größe schneller erreicht wird. Im Folgenden soll neben der Überprüfung dieser Vermutung auch der Aufwand gemessen werden.

Der Aufwand zum Wechseln der Modi wird anhand der Anzahl der versendeten Pakete auf der MAC-Schicht gemessen. Die Effektivität wird anhand der Zeit gemessen, wie lange sich Knoten in einem falschen Modus befinden. Wie bereits oben erläutert, befinden sich in der Simulation 100 zufällig verteilte Knoten in einem Areal von 1000 m x 1000 m. Die Kommunikationsreichweite ist auf 150 m eingestellt. Während einer Simulationszeit von 20 Sekunden initiieren 10 zufällig ausgewählte Knoten zu einem ebenfalls zufällig gewählten Zeitpunkt eine Modus-Anforderung zwischen Modus 0 und 2. Insgesamt wurden jeweils 60 Experimente durchgeführt und gemittelt.

In Abb. 4.12 ist die durchschnittliche Anzahl der Pakete auf MAC-Schicht dargestellt, die nötig ist, um die Modus-Wechsel durchzuführen. Die blau-gepunktete Kurve „Cache 0“ steht für das Modus-Wechsel-Protokoll *Zeroknowledge* (d. h. Cache-Größe 0), die grün-gestrichelte Kurve „Cache 1“ für den *Multiknowledge* bei Cache-Größe 1, und schließlich die rot-durchgezogene Kurve für *Multiknowledge* bei Cache-Größe 2.

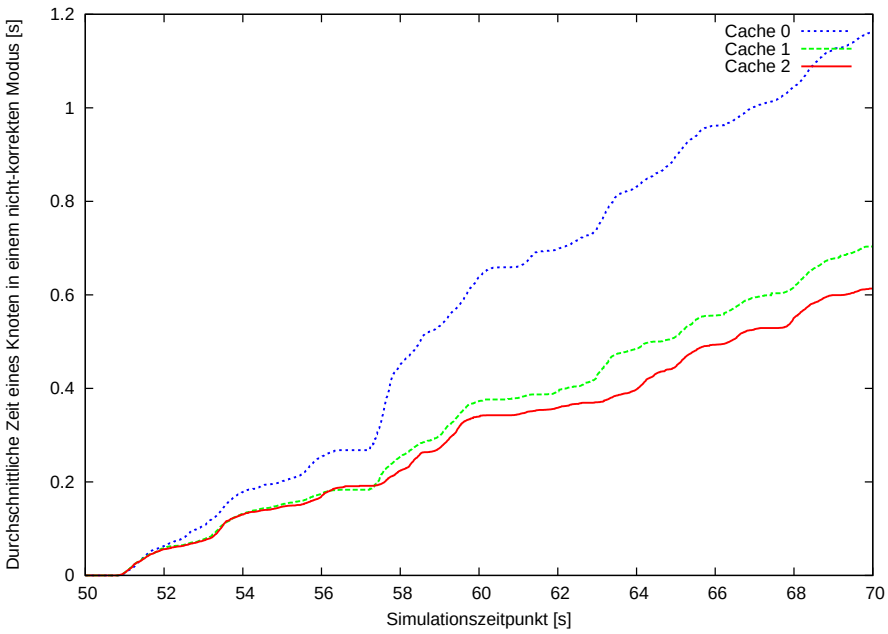

Abbildung 4.13: Kumulierte Zeit aller Knoten, die sich im falschen Modus befinden (bei 10 Modus-Anforderungen während der Simulationszeit); für die Simulationsparameter siehe auch Tab. B.4 im Anhang

Schon nach 20 Sekunden Simulationsdauer (von Simulationszeit 50 bis 70) benötigt „Cache 1“ nur etwa halb soviele MAC-Pakete, um die Modus-Wechsel durchzuführen, wie „Cache 0“. Durch Ausnutzung der Zusatzinformation werden sehr viele unnötige Modus-Anforderungspakete vermieden, die insbesondere beim Auslaufen einer Modus-Anforderung bei „Cache 0“ versendet werden (vgl. dazu die obigen Betrachtungen eines Einzelexperiments).

Etwas überraschend verläuft die Kurve bei „Cache 2“. Im Vergleich zu „Cache 1“ werden stets mehr Pakete verschickt: Die durch den weiteren Cache-Platz zusätzlich verschickten Modus-Anforderungspakete überwiegen die Einsparungen durch unnötige Modus-Anforderungspakete beim Auslaufen einer Modus-Anforderung. Der Vergleich zu „Cache 0“ zeigt, dass zu Beginn der Simulation mehr Pakete verschickt werden, um die Cache-Plätze zu füllen. Nach der initialen Füllphase (bis etwa Simulationszeit 53 Sekunden) verläuft die Kurve wegen eingesparter Modus-Anforderungspakete flacher, so dass ab etwa 64 Sekunden insgesamt weniger Pakete verschickt werden als bei „Cache 0“ (wenn auch deutlich mehr als bei „Cache 1“).

In Abb. 4.13 ist zu sehen, wieviel Zeit die Knoten benötigen, den korrekten Modus einzunehmen. Auf der y-Achse ist zu diesem Zweck die durchschnittliche, auf einen Knoten normierte Zeit aufgetragen, wie lange sich die Knoten in einem falschen Modus befinden. Hier zeigen sich deutlich die Vorteile des Cachings. Über die gesamte Simulationszeit von 20 Sekunden ist die durchschnittliche Zeit eines Knotens im falschen Modus bei Cache-Größe 2 wesentlich geringer als bei Cache-Größe 0. Gegenüber „Cache 0“ kann „Cache 2“ die inkonsistente Zeit fast halbieren und liegt gegenüber „Cache 1“ ebenfalls deutlich besser. Am Ende der Simulationszeit summiert sich die durchschnittliche Zeit pro Knoten in inkorrektem Modus auf etwa 0,6 s bei „Cache 2“, etwa 0,7 s bei „Cache 1“ und etwa 1,1 s bei „Cache 0“.

Insgesamt zeigt sich, dass bei 10 Modus-Anforderungen in 20 Sekunden und 3 zur Verfügung stehenden Modi „Cache 0“ sowohl bei der Anzahl der versendeten Pakete wie auch bei der Zeit, den korrekten Modus einzunehmen, im Vergleich am schlechtesten abschneidet. „Cache 1“ liegt unter den genannten Bedingungen bei der Anzahl der gesendeten Pakete vorne, während „Cache 2“ die kürzeste Zeit für den Wechsel in den korrekten Modus aufweist.

4.9.2 Untersuchung der Modus-Wechsel-Protokolle mit verschiedenen Cache-Größen und Modus-Anzahl

In der vorangegangen Untersuchung wurde das Verhalten der Modus-Wechsel-Protokolle für die Cache-Größen von 0 bis 2, bei 10 Modus-Anforderungen in 20 Sekunden, und bei drei zur Verfügung stehenden Modi analysiert. Im Folgenden sollen die Auswirkungen bei systematisch geänderten Ausgangsbedingungen untersucht werden. Insbesondere wird die Frequenz der Modus-Anforderungen erhöht und sowohl die Anzahl der Cache-Größe wie auch die Anzahl der Modi erhöht. Die übrigen Simulationsbedingungen entsprechen den zuvor angegebenen (100 Knoten, 1000 m x 1000 m Areal, 150 m Reichweite). Die gesamte Simulationszeit liegt bei sämtlichen folgenden Simulationen bei 100 Sekunden. Die Gültigkeitsdauer der Modus-Anforderungen ist 10 Sekunden.

In Abb. 4.14 wird die Zeit der Knoten, die sie sich im falschen Modus befinden, bei 10 Modus-Anforderungen dargestellt. Auf der x-Achse wird die Zahl der zur Verfügung stehenden Modi von 2 bis 10 in Zweierschritten erhöht. Die prinzipielle Aussage der Beobachtung im letzten Abschnitt bestätigt sich. Im Einklang mit den vorigen Ergebnissen (siehe Abb. 4.13) sinkt unabhängig von der Anzahl der zur Verfügung stehenden Modi die Zeit eines Knoten im falschen Modus mit zunehmender Cache-Größe.

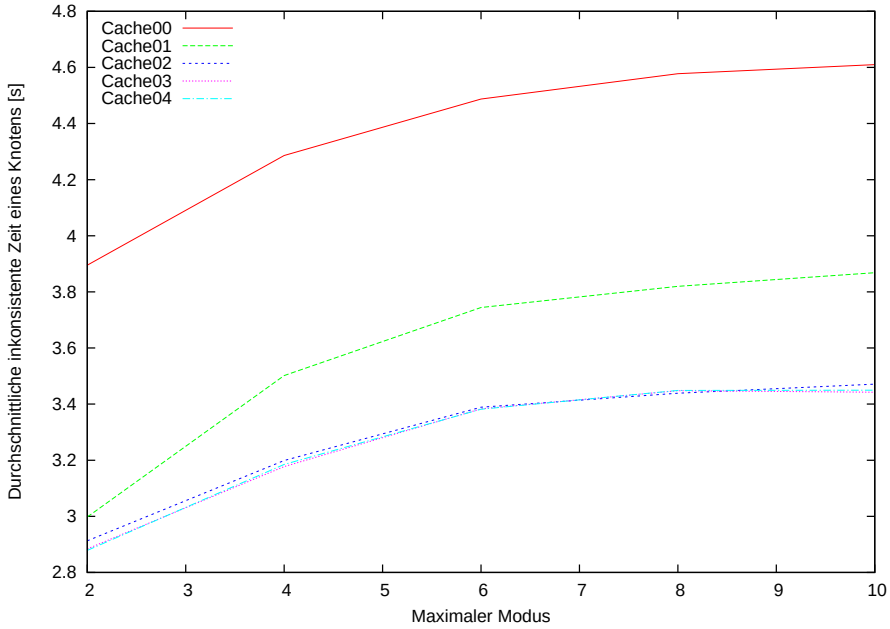

Abbildung 4.14: Durchschnittliche Zeit eines Knoten im falschen Modus bei 10 Modus-Anforderungen; für die Simulationsparameter siehe auch Tab. B.5 im Anhang

Allerdings gilt dies nur bis zu einer Cache-Größe von 2. Eine weitere Erhöhung der Cache-Größe ergibt nur einen minimalen Zugewinn an Konsistenz. Dies liegt zum einen daran, dass äußerst selten eine dritte oder vierte zusätzlich gespeicherte Modus-Anforderung tatsächlich wirksam wird. Zum anderen sind bei der relativ geringen Rate an Modus-Anforderungen (durchschnittlich eine Modus-Anforderung in 15 Sekunden bei einer Gültigkeitsdauer von 10 Sekunden) nur selten tatsächlich mehr als 2 Modus-Anforderungen gleichzeitig vorhanden.

Ein Vergleich der für die jeweiligen Cache-Größen benötigten MAC-Pakete ist in Abb. 4.15 zu sehen. Auch hier wiederholen sich die Ergebnisse des vorhergehenden Abschnitts. Unabhängig von der Anzahl der zur Verfügung stehenden Modi benötigt „Cache00“ die meisten Pakete, während „Cache01“ die wenigsten versendeten Pakete aufweist. Tendenziell steigt die Zahl der gesendeten Pakete mit zunehmender Cache-Größe, wobei ab einer Cache-Größe von 3 der Unterschied nur noch marginal ist.

Um das Verhalten der Modus-Wechsel-Protokolle bei sehr vielen Modus-Anforderungen zu untersuchen, wird das Experiment mit 50 Modus-Anforderungen wiederholt. In Abb. 4.16 ist Zeit der Knoten aufgetragen, in der sie sich im falschen Modus befin-

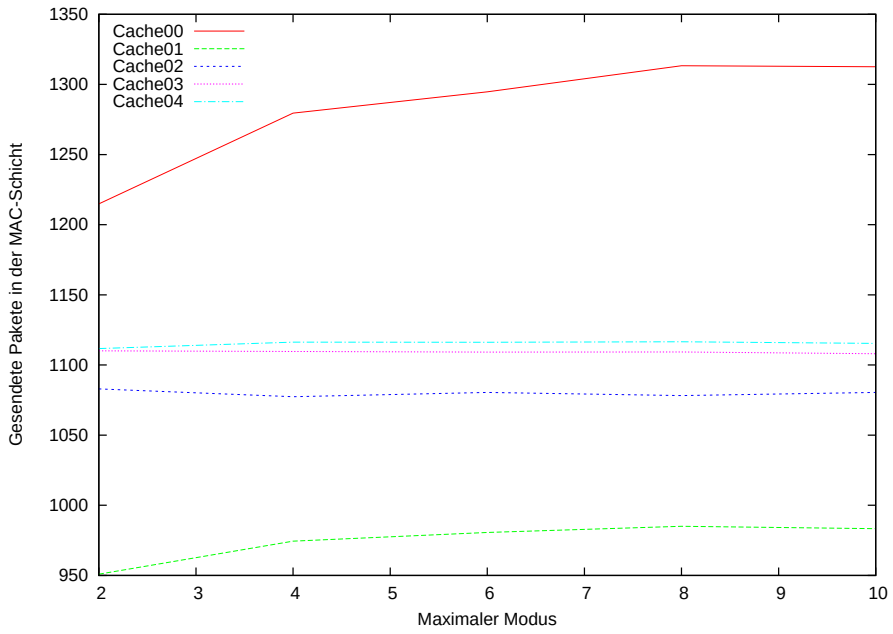

Abbildung 4.15: Zahl der versendeten MAC-Pakete bei 10 Modus-Anforderungen; für die Simulationsparameter siehe auch Tab. B.5 im Anhang

den. Qualitativ ergibt sich das gleiche Ergebnis wie bei 10 Modus-Anforderungen: Auch hier ist die inkonsistente Zeit bei „Cache00“ am größten, gefolgt von „Cache01“. Ab der Cache-Größe 2 sind kaum noch Unterschiede festzustellen.

Im quantitativen Vergleich zu 10 Modus-Anforderungen ist außer bei „Cache00“ insgesamt die Zeit im falschen Modus wesentlich geringer. Durch die hohe Anzahl der Modus-Anforderungen ist der global korrekte Modus über längere Zeit gleichbleibend als bei weniger Modus-Anforderungen, da zwei verschiedene Knoten zu sich überlappenden Zeitabschnitten die gleiche Modus-Anforderung haben können. Ist kein Cache-Platz vorhanden, bietet dies keinen Vorteil, da jedes Auslaufen einer Gültigkeit ein Verbreiten des Standard-Modus 0 zur Folge hat. Ist dagegen mindestens ein Cache-Platz vorhanden, steigt die Wahrscheinlichkeit, dass der Übergang ohne inkorrekte Zwischen-Modi direkt in den korrekten Netzmodus erfolgt.

Bei der Zahl der versendeten MAC-Pakete ergibt sich ein leicht geändertes Bild bei 50 Modus-Anforderungen im Vergleich zu 10 Modus-Anforderungen. In Abb. 4.17 sind die Ergebnisse dargestellt. Zwar ist auch hier die Anzahl bei „Cache00“ am größten und bei „Cache01“ am geringsten, allerdings steigt die Anzahl mit größer werdendem Cache auch jenseits von „Cache02“. Dies liegt daran, dass bei der hohen Rate an Modus-

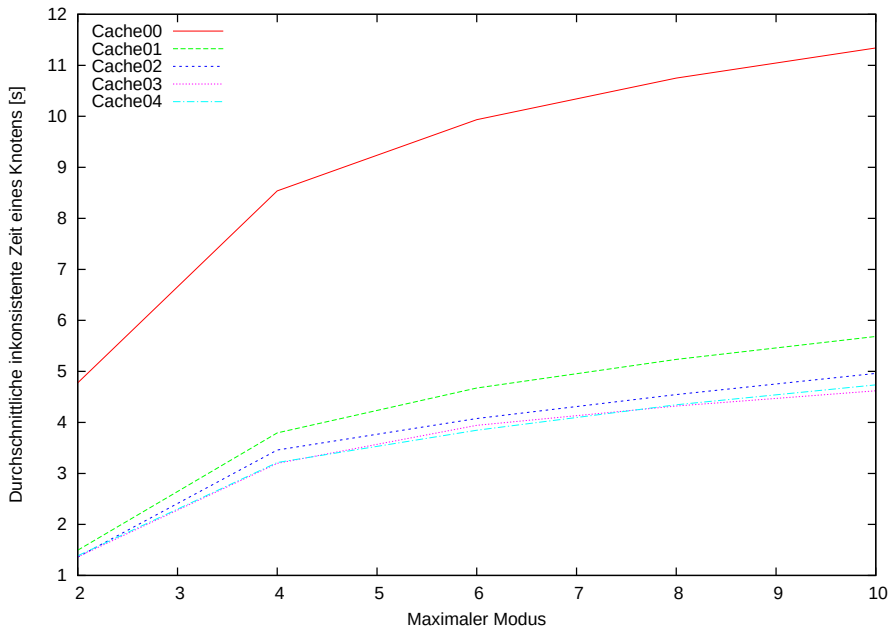

Abbildung 4.16: Durchschnittliche Zeit eines Knoten im falschen Modus bei 50 Modus-Anforderungen; für die Simulationsparameter siehe auch Tab. B.6 im Anhang

Anforderungen (durchschnittlich eine Modus-Anforderung in 3 Sekunden bei einer Gültigkeitsdauer von 10 Sekunden) der Cache immer aufgefüllt werden kann und somit die Modus-Anforderungspakete bei nahezu jeder Modus-Anforderung verbreitet werden.

Zusammenfassung

In den Simulationsergebnissen wird deutlich, dass die Wahl der Cache-Größe wesentlich den Zusatzaufwand für den Modus-Wechsel und die Konsistenz des Netzes beeinflusst, während die Anzahl zur Verfügung stehender Modi kaum Einfluss hat. Eine Cache-Größe von 1 hat sowohl bei gelegentlichen als auch bei sehr häufigen Modus-Wechseln den geringsten Zusatzaufwand. Eine Cache-Größe von 2 führt zu einer erheblichen Verbesserung der Konsistenz, während ein noch größerer Cache nicht zu einer signifikanten Verbesserung führt. Allenfalls bei sehr häufigen Modus-Wechseln kann eine Cache-Größe von 3 zu einer nochmals leicht verbesserten Konsistenz führen.

Für konkrete Anwendungen sollte deswegen je nach Anforderung und erwarteter Häufigkeit von Modus-Wechseln eine Cache-Größe von 1 bis 3 gewählt werden. Größere

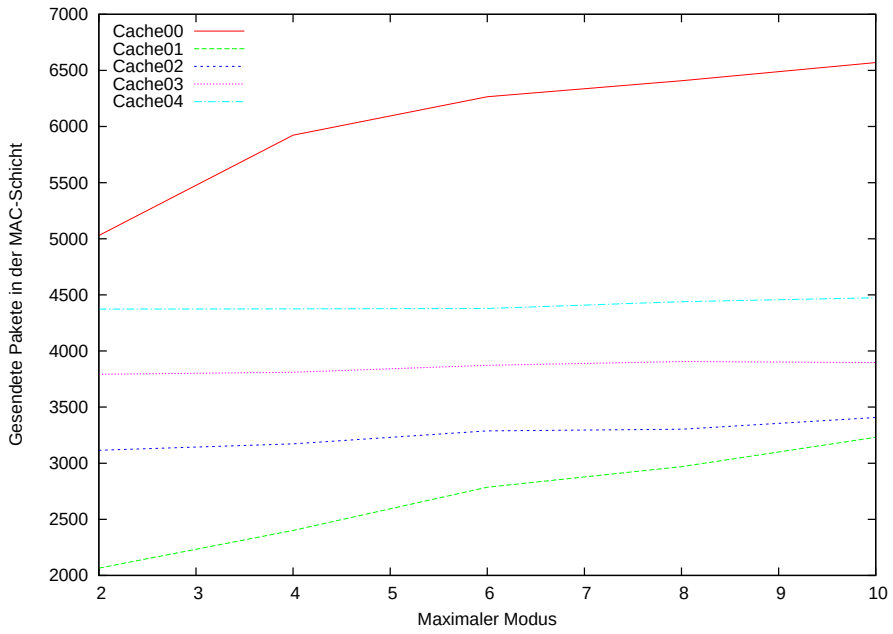

Abbildung 4.17: Zahl der versendeten MAC-Pakete bei 50 Modus-Anforderungen; für die Simulationsparameter siehe auch Tab. B.6 im Anhang

Cache-Größen sind aufgrund der hier simulativ erworbenen Erkenntnisse unnötig. Die in den zusätzlichen Cache-Plätzen vorgehaltenen Modus-Anforderungen führen sehr selten zu einem tatsächlichen Modus-Wechsel, da sie zumindest bei einer Gleichverteilung sehr oft von höherprioren Modus-Anforderungen verdrängt werden. Die Begrenzung der Cache-Größe wirkt sich ohne sonstige Nachteile positiv auf den Speicherverbrauch aus und verringert die Protokollkomplexität sowie den Zusatzaufwand für Kommunikation.

4.9.3 Untersuchung der Modus-Implementierungen

Nachdem der Modus-Wechsel ohne den Einfluss einer laufenden Modus-Implementierung untersucht wurde, soll im nächsten Abschnitt nachgewiesen werden, dass die Umschaltung des Modus effektiv und effizient auch bei einer laufenden Modus-Implementierung möglich ist. Dazu werden in diesem Abschnitt drei beispielhafte Modus-Implementierungen vorgestellt. Diese dienen nicht vordergründig dazu, das Modusziel optimal umzusetzen, sondern vielmehr exemplarisch und nur im gegenseitigen Vergleich das jeweilige Modusziel zu erreichen. Dieser gegenseitige Vergleich soll im Folgenden simulativ durchgeführt werden.

Zu diesem Zweck wurden die in Abschnitt 4.4 vorgestellten Modus-Implementierungen für einen energiesparenden, schnellen und robusten Modus in ns2 implementiert. Dazu werden die Knoten zu Kommunikationsgruppen zusammengefasst. Im Folgenden sind 100 Knoten auf einem Areal von 1000 m x 1000 m zufällig platziert. Das quadratische Gesamtareal wird in 4x4 Quadrate von jeweils 250 m Kantenlänge aufgeteilt und die darinliegenden Knoten einer Kommunikationsgruppe zugeordnet. Die angenommene Aufgabe einer Kommunikationsgruppe ist es, jede Sekunde ein Datum an einen zufälligen, aber für alle Teilnehmer der Kommunikationsgruppe gleichen Senkeknoten zu senden. Im energiesparenden Modus senden die Teilnehmer ihr Datum an einen festen Knoten der Gruppe, der alle Daten sammelt und nach 20 Sekunden in einem einzigen Paket an den Senkeknoten sendet. Im schnellen Modus senden die Teilnehmer ihre Daten an den Senkeknoten sofort und direkt. Im robusten Modus senden die Teilnehmer ihre Daten jeweils auf zwei Wegen. Zum Einen wird das Paket auf dem vom Routingprotokoll gewählten Weg „direkt“ gesendet, zum Zweiten wird das Paket in zwei Etappen über einen von jedem Knoten zufällig aus dem gesamten Netz gewählten anderen Knoten gesendet. Zum Senden der Pakete wird die ns2-eigene Standard-Implementierung von AODV verwendet. Die Reichweite ist auf 150 m festgelegt. Die Simulationszeit beträgt 100 Sekunden.

In den Simulationen werden die Anzahl, die Verzögerung und die Ankunftswahrscheinlichkeit der versendeten Pakete untersucht, um die Effizienz des energiesparenden, schnellen und robusten Modus zu untersuchen.

Die Anzahl der versendeten Pakete bei den jeweiligen Modus-Implementierungen wird in Abb. 4.18 verglichen. Auf der y-Achse ist die Anzahl der versendeten Pakete aufgetragen. Auf der x-Achse wird die Ausfallwahrscheinlichkeit der Knoten variiert. Ein ausfallender Knoten fällt zu einem gleichverteilt zufälligen Zeitpunkt während der Simulationszeit aus und bleibt bis zum Ende der Simulation in diesem Zustand. In Abb. 4.18 ist der energiesparende Modus als blau-gepunktete, der schnelle Modus als grün-gestrichelte und der robuste Modus als rot-durchgezogene Kurve dargestellt. Die Anzahl der versendeten MAC-Pakete ist im energiesparenden Modus bei allen Knotenausfallwahrscheinlichkeiten durchgängig geringer als in den anderen beiden Modi.

Im direkten Vergleich der Modi ist somit die prinzipielle Effektivität der exemplarischen energiesparenden Modus-Implementierung bezüglich der Energieeffizienz nachgewiesen.

Um die Effektivität des schnellen Modus nachzuweisen, wird in Abb. 4.19 die durchschnittliche Verzögerung der Pakete zwischen Senden und Empfangen verglichen. Hier

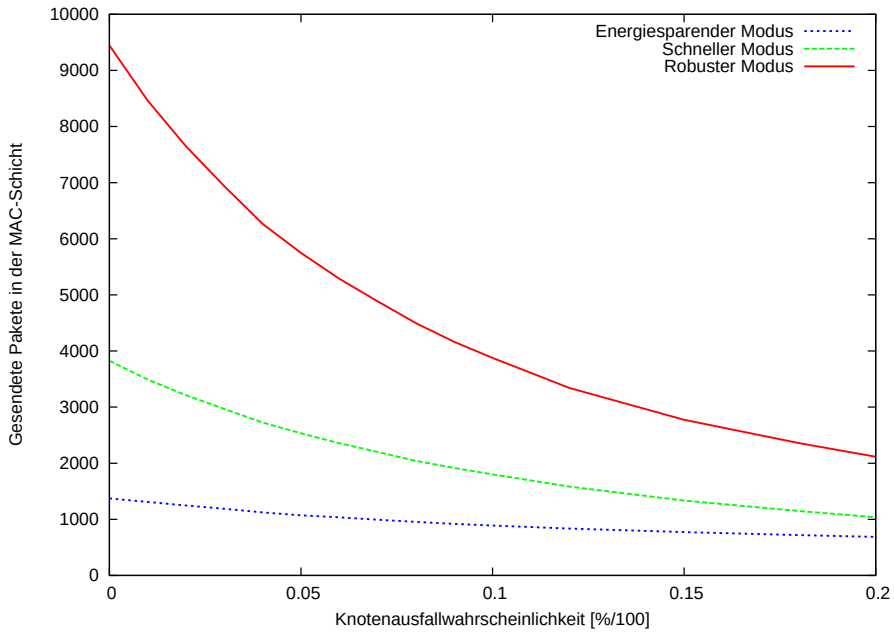

Abbildung 4.18: Vergleich der Modus-Implementierungen anhand der Anzahl der versendeten Pakete; für die Simulationsparameter siehe auch Tab. B.7 im Anhang

zeichnet sich wiederum erwartungsgemäß der schnelle Modus aus, der durchgängig eine geringere Verzögerung aufweist als die anderen Modi.

Im direkten Vergleich der Modi ist somit die prinzipielle Effektivität der exemplarischen schnellen Modus-Implementierung bezüglich der Latenz nachgewiesen.

In Abb. 4.20 ist die Anzahl der empfangenen Pakete zu sehen. Bei einer Ausfallwahrscheinlichkeit von 0 Prozent erreichen alle Pakete den festgelegten Senkeknoten. Fehler auf der MAC-Ebene, wie z. B. Kollisionen, werden durch AODV vollständig abgefangen. Bei Ausfallwahrscheinlichkeiten größer 0 Prozent werden im robusten Modus durchgängig mehr Pakete empfangen als in den anderen beiden Modi.

Im direkten Vergleich der Modi ist somit die prinzipielle Effektivität der exemplarischen robusten Modus-Implementierung bezüglich der Robustheit nachgewiesen.

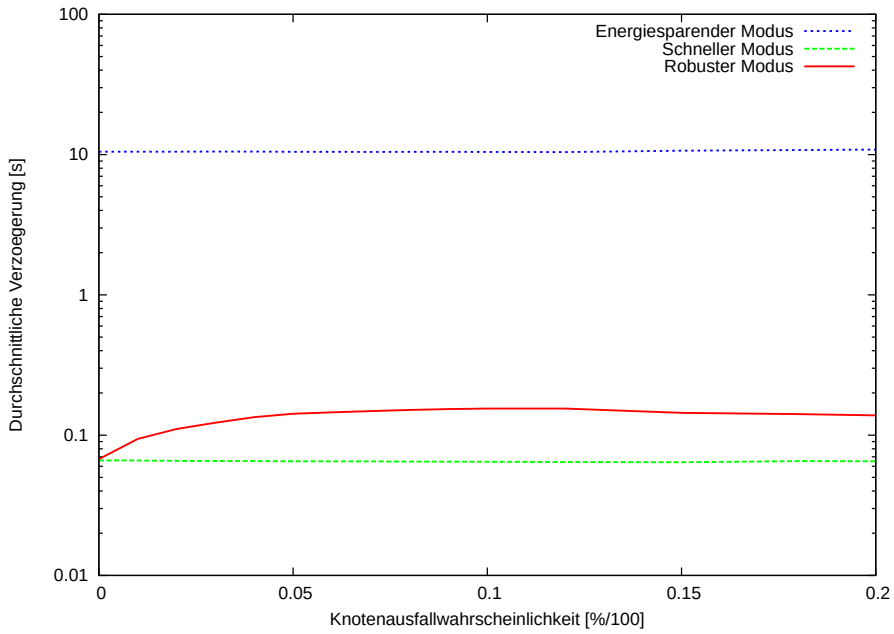

Abbildung 4.19: Vergleich der Modus-Implementierungen anhand der durchschnittlichen Verzögerung versendeter Pakete; für die Simulationsparameter siehe auch Tab. B.7 im Anhang

4.9.4 Untersuchung der Modus-Wechsel-Protokolle bei laufenden Modus-Implementierungen

Nachdem die prinzipielle Eignung der exemplarischen Modus-Implementierungen im vorangegangen Abschnitt nachgewiesen worden ist, soll hier der Modus-Wechsel zwischen diesen dreien untersucht werden. In einzelnen Experimenten wird der Nachweis geführt, dass durch den Zusatzaufwand des Modus-Wechsels nicht die eigentliche Intention des Zielmodus konterkariert wird.

Der Zusatzaufwand besteht im für den Modus-Wechsel notwendigen Fluten des Netzes und existiert unabhängig vom konkreten Zeitpunkt des Modus-Wechsels. Eine Variation des Zeitpunktes ergibt deswegen keine geänderten Resultate. Somit wird der Nachweis in repräsentativ geltenden Einzelexperimenten geführt.

Die angenommene Aufgabe und damit der Aufbau der Experiments entspricht den im vorangegangenen Abschnitt erläuterten Simulationsparametern.

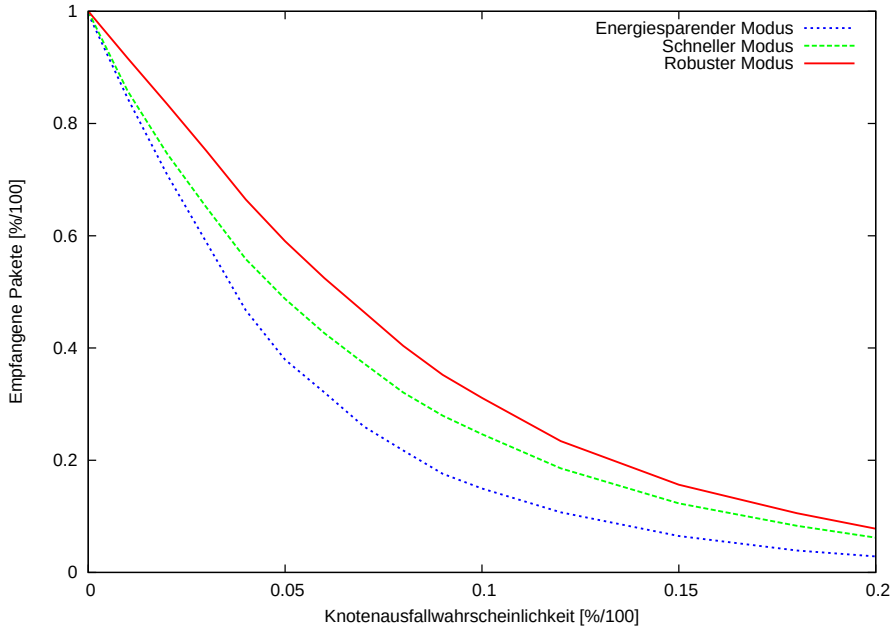

Abbildung 4.20: Vergleich der Modus-Implementierungen anhand der Anzahl der empfangenen Pakete; für die Simulationsparameter siehe auch Tab. B.7 im Anhang

In Abb. 4.21 ist der Vergleich zwischen dem schnellen Modus in der blau-gepunkteten Kurve und dem energiesparenden Modus in der grün-gestrichelten Kurve dargestellt. Auf der y-Achse sind die versendeten MAC-Pakete aufgetragen; auf der x-Achse die Zeit. Bis zum Zeitpunkt 240 Sekunden befinden sich beide Anwendungen (in separat ausgeführten Simulationen) im schnellen Modus. Zum Zeitpunkt 240 Sekunden findet bei der grün-gestrichelten Kurve die Umschaltung auf den energiesparenden Modus statt. Der deutliche Anstieg der versendeten Pakete rührt dabei von den für den Modus-Wechsel versendeten Paketen. Im Anschluss werden im Vergleich zum schnellen Modus deutlich weniger Pakete versendet. Die Knoten senden in diesem Modus ihre Pakete nur noch einem näherliegenden, für alle Teilnehmer der Kommunikationspartner festen Knoten. Dieser sendet dann alle 20 Sekunden die aggregierten Ergebnisse an den definierten Senkeknoten, was an den periodisch auftretenden Stufen bei der Anzahl der versendeten Pakete deutlich erkennbar ist. Nach etwas weniger als 30 Sekunden zum Zeitpunkt 270 Sekunden unterschreitet die Anzahl trotz dem für den Modus-Wechsel benötigten Zusatzaufwand dauerhaft die Anzahl der versendeten Pakete beim schnellen Modus.

In Abb. 4.22 ist der Vergleich zwischen dem schnellen Modus mit rot-durchgezogen Balken und dem energiesparenden Modus mit grün-gestrichelten Balken dargestellt.

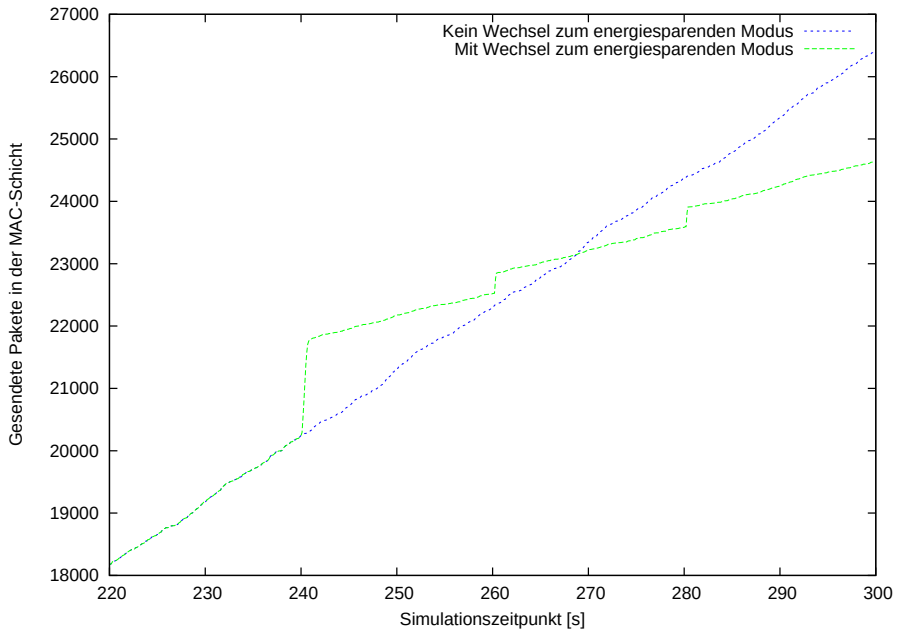

Abbildung 4.21: Vergleich der versendeten Pakete mit und ohne Wechsel auf energiesparenden Modus; für die Simulationsparameter siehe auch Tab. B.8 im Anhang

Auf der x-Achse ist die Zeit aufgetragen; auf der y-Achse die durchschnittliche Verzögerung der zu diesem Zeitpunkt eingetroffenen Pakete. Bis zum Zeitpunkt 250 Sekunden befinden sich beide Anwendungen (in separat ausgeführten Simulationen) im energiesparenden Modus. Zum Zeitpunkt 250 Sekunden findet in der grün-gestrichelten Simulation ein Wechsel auf den schnellen Modus statt. Ab diesem Zeitpunkt erreichen die Pakete mit einer durchschnittlichen Verzögerung von etwa 0,1 Sekunden den Senkeknoten anstatt mit einer durchschnittlichen Verzögerung von etwa 10 Sekunden beim energiesparenden Modus. Aufgrund der hier implementierten Umsetzung des Modus-Wechsels, werden zum Zeitpunkt des Modus-Wechsels die noch im alten, energiesparenden Modus versendeten Pakete nicht unmittelbar vom aggregierenden Knoten weitergesendet. Stattdessen gilt für diese Pakete nach wie vor das Ziel, Energie zu sparen. Deswegen werden zum Zeitpunkt 260 Sekunden noch ein letztes Mal Pakete aggregiert versendet, die aus der Zeit 240 bis 250 Sekunden mit dem energiesparenden Modus versendet worden sind. Die durchschnittliche Verzögerung aller zu diesem Zeitpunkt ankommenden Pakete liegt deswegen mit etwa 12 Sekunden leicht höher als bei der konstant im energiesparenden Modus durchlaufenden Vergleichssimulation.

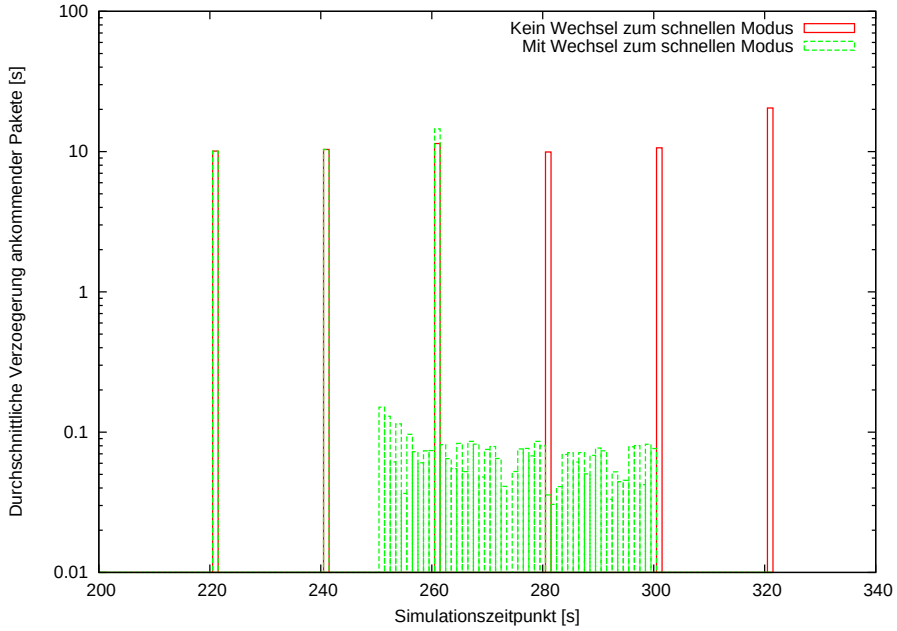

Abbildung 4.22: Vergleich der durchschnittlichen Verzögerung mit und ohne Wechsel auf schnellen Modus; für die Simulationsparameter siehe auch Tab. B.9 im Anhang

In Abb. 4.23 ist der Vergleich zwischen dem energiesparenden Modus in der blau-gepunkteten Kurve und dem robusten Modus in der grün-gestrichelten Kurve dargestellt. Auf der x-Achse ist die Zeit aufgetragen, auf der y-Achse der durchschnittliche Anteil der empfangenen Pakete. Während der gesamten Simulationszeit fallen durchschnittlich 5 Prozent der Knoten für 1 Sekunde aus. Bis zum Zeitpunkt 250 Sekunden befinden sich beide Anwendungen (in separat ausgeführten Simulationen) im energiesparenden Modus. Zum Zeitpunkt 250 findet in der grün-gestrichelten Simulation ein Wechsel auf den robusten Modus statt. Ab diesem Zeitpunkt steigt der Anteil der empfangenen Pakete deutlich auf ca. 33 Prozent an, während der Anteil beim energiesparenden Modus bei ca. 15 Prozent verbleibt.

4.9.5 Zusammenfassung der Evaluation der Modus-Wechsel-Protokolle

In der vorangegangenen Evaluation der Modus-Wechsel-Protokolle wurde das Verhalten mit verschiedenen Cache-Größen untersucht und in einzelnen Experimenten die unterschiedliche Reaktion bei Modus-Wechseln aufgezeigt und erklärt. Die wichtigsten

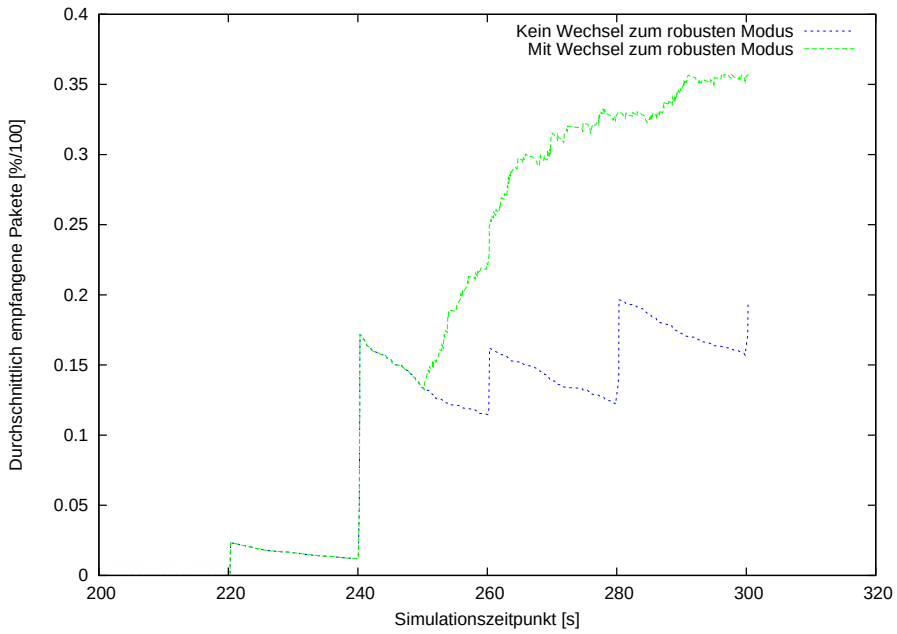

Abbildung 4.23: Vergleich der empfangenen Pakete mit und ohne Wechsel auf robusten Modus; für die Simulationsparameter siehe auch Tab. B.10 im Anhang

Kenngrößen, d. h. der Aufwand und die Effektivität, wurden mit variierenden Simulationsparametern der Cache-Größe, Häufigkeit des Modus-Wechsels und Anzahl der zur Verfügung stehenden Modi untersucht. Bei allen untersuchten Simulationskonstellationen war der Zusatzaufwand des Modus-Wechsel-Protokolls mit Cache-Größe 1 am geringsten. Die Effektivität, gemessen an der durchschnittlichen Zeit der Knoten im falschen Modus, war durchgängig bei einer Cache-Größe 2 nahe des Maximums. Die Effektivität nahm bei Cache-Größe 3 oder 4 nur in geringem Maße zu. Im Abschluss wurde nachgewiesen, dass ein Modus-Wechsel zwischen den exemplarischen Modus-Implementierungen für energiesparenden, schnellen und robusten Modus auch unter Beachtung des Zusatzaufwands der Modus-Wechsel-Protokolle die gewünschte Kommunikationsoptimierung erreicht.

Für die Verwendung in einem realen Sensornetz können alle gezeigten Modus-Wechsel-Protokolle verwendet werden. Das einfachste Modus-Wechsel-Protokoll *Zeroknowledge* (in den Experimenten als Cache 0 bezeichnet) bietet sich für Sensornetze an, deren Sensorknoten insbesondere in der Rechenkapazität und Speicherkapazität beschränkt sind, da dieser sich durch das einfachste Regelwerk auszeichnet und keinen weiteren Speicherbedarf durch Cache-Plätze benötigt. Sind die Sensorknoten dagegen fähig, auch ein komplexeres Protokoll für den Modus-Wechsel auszuführen, und

stellen mehr Speicherkapazität bereit, kann das Modus-Wechsel-Protokoll *Multiknowledge* eingesetzt werden. Dabei kann durch die Wahl der Cache-Größe entweder auf den Speicherbedarf oder auf die Schnelligkeit für den Modus-Wechsel optimiert werden. Die Ergebnisse der Evaluation legen nahe, dass eine größere Cache-Größe ab einer Anzahl von 3 nicht mehr für einen effizienteren (schnelleren) Modus-Wechsel sorgt. Dies liegt insbesondere daran, dass bei einer zufälligen Wahl des Modus die zusätzlich im Cache gespeicherten Modus-Anforderungen mit geringerer Priorität nicht mehr relevant sind. Sie werden durch zeitlich nachfolgende Modus-Anforderungen mit höherer Priorität obsolet. Werden in einem Sensornetz zeitlich zufällig gleichverteilte Modus-Anforderungen erwartet, kann somit die Cache-Größe auf maximal 3 beschränkt werden.

Die Beschränkung der Cache-Größe verringert damit den Speicherverbrauch, die Protokollkomplexität und den Zusatzaufwand für Kommunikation, ohne dabei Nachteile eingehen zu müssen.

Der grundlegende Aufbau des Modus-Rahmenwerks wurde in [PHWZ05] veröffentlicht. Für die im Zuge dieser Arbeit entwickelte Architektur für anwendungsgesteuerte Kommunikationsoptimierung, die Modus-Wechsel-Protokolle und deren Zusammenhänge mit Modus-Implementierungen wurde zusammen mit DoCoMo Ltd. ein Antrag auf ein europäisches Patent gestellt und erteilt [HPW+05].

5 Zusammenfassung und Ausblick

In dieser Arbeit wurden PanTalassa zur dienst- und datenorientierten Programmierung und das Modus-Rahmenwerk zur anwendungsgesteuerten Kommunikationsoptimierung von Sensornetzen entwickelt. Mit diesen beiden Entwicklungen gelingt es, Sensornetze für unterschiedliche Anwendungen zu nutzen und an unterschiedliche Situationen anzupassen.

5.1 Zusammenfassung

PanTalassa erlaubt die Implementierung von verteilten Anwendungen mit Diensten. PanTalassa-Anwendungen werden über drei separate Teile definiert und implementiert: Datenfluss, Datenverarbeitung, Kontrollfluss. Der Datenfluss wird über primitive Talassa-Dienste und primitive Pan-Daten definiert, die netzinterne Datenverarbeitung über abgeleitete Pan-Daten, und der Kontrollfluss über komponierende Talassa-Dienste. Dies ermöglicht eine übersichtliche, graphisch visualisierbare und strukturierte Anwendungsentwicklung.

Mit einer Beschränkung auf fünf grundlegende Diensttypen lässt sich PanTalassa effizient auf ressourcenarmen Sensorknoten implementieren. Trotz der Beschränkung kann durch den dargelegten Nachweis der Turingmächtigkeit garantiert werden, dass PanTalassa für beliebige Anwendungen eingesetzt werden kann.

Durch die geringe Größe der Dienst- und Datendefinitionen können verteilte Anwendungen einfach über die Luftschnittstelle an das Sensornetz übertragen und von den Sensorknoten effizient ausgeführt werden.

Weitgehende Freiheitsgrade bei der Wahl des Ausführungsorts, insbesondere bei den komponierenden Diensten, können dazu genutzt werden, die Zuordnung der Dienste zu Sensorknoten autonom und situationsangepasst im laufenden Betrieb zu ändern. Es wurde nachgewiesen, dass durch die in dieser Arbeit entwickelten Verfahren zur autono-

men Dienst-Knoten-Zuordnung die Lebenszeit einer Sensornetzanwendung signifikant gesteigert wird, indem Dienste während der Laufzeit energiearme Knoten verlassen und auf andere geeignete Knoten wechseln. Auch die im gesamten Sensornetz benötigte Energie wurde auf diese Weise erheblich verringert.

Die Definition von PAN-Daten ermöglicht, Sensorwerte netzintern zu aggregieren, zu reduzieren oder benutzerdefiniert zu verarbeiten. PAN-Daten können auf einfache Weise von mit TALASSA definierten Diensten genutzt werden, womit die Vorteile von datenorientierten und dienstorientierten Sensornetzen vereint werden konnten.

Die Umsetzung von PANTALASSA für reale Sensornetze basiert auf der Implementierung der virtuellen Maschine PANTALASSAVM, die die Ausführung der TALASSA-Dienste und Verarbeitung der PAN-Daten sicherstellt. Virtuelle Maschinen garantieren eine weitgehende Unabhängigkeit von speziellen Hardwareplattformen und Betriebssystemen. Somit erlaubt auch PANTALASSA den Einsatz von Sensornetzen mit heterogenen Sensorknoten bezüglich Sensorknoten-Hardwareplattform und -Betriebssystem.

Die PANTALASSAVM wurde auf einer Sensorknoten-Plattform von Crossbow implementiert. Auf Basis dieser Implementierung wurde ein dienstorientiertes intelligentes Gewächshaus modelliert und umgesetzt, das erfolgreich auf einer internationalen Konferenz als Demonstrator vorgeführt wurde.

Das in dieser Arbeit entwickelte Modus-Rahmenwerk ermöglicht die Festlegung verschiedener Modi (d. h. verschiedener Optimierungsziele für die Kommunikation), die Anwendungen in einem Sensornetz setzen beziehungsweise wechseln können, um den Kommunikationsanforderungen in einer bestimmten Situation gerecht zu werden. Die Entscheidung des zu benutzenden Modus und der Zeitpunkt des Wechsels obliegt der Anwendung und kann von jedem beliebigen Knoten der verteilten Anwendung ausgelöst werden. Auf diese Weise kann eine verteilte Anwendung ein Sensornetz für verschiedene Zwecke nutzen, wie z. B. für energiesparendes Datensammeln aber auch für die schnelle Weiterleitung von Alarmmeldungen.

Für den Modus-Wechsel bietet das Modus-Rahmenwerk ein effiziente Protokoll, die bewiesenermaßen konsistente und korrekte, netzweite Modus-Wechsel ermöglichten. Je nach Anforderung bietet das Modus-Rahmenwerk Protokolle, die auf besonders schnelle netzweite Modus-Wechsel, Speicherplatzverbrauch oder Einfachheit der Implementierung optimiert werden können. Das Modus-Rahmenwerk und die Modus-Wechsel-Protokolle wurden auf europäischer Ebene patentrechtlich geschützt.

In Kombination bieten das Modus-Rahmenwerk und PanTalassa eine erhebliche Ausweitung der Anwendungsmöglichkeiten von Sensornetzen. Ohne Änderung der Infrastruktur, der Hardware oder des Betriebssystems der einzelnen Sensorknoten ist es möglich, neue Anwendungen auszubringen, Kommunikation bestehender Anwendungen zu ändern und die Kommunikation der jeweiligen Situation anzupassen. Im Gegensatz zu bisherigen Ansätzen ist es somit möglich, ein Sensornetz für völlig verschiedene Anwendungen parallel zu nutzen. Gleichzeitig wird die Lebenszeit eines Sensornetzes bedeutend verlängert, indem anwendungsgesteuert das Kommunikationsverhalten an die Situation angepasst werden kann und verteilte Anwendungen autonom und dynamisch eine kommunikationseffiziente Dienst-Knoten-Zuordnung herstellen.

5.2 Ausblick

Das Modus-Rahmenwerk schlägt momentan drei allgemein nutzbare Modi vor: energiesparend, schnell und robust. Diese Modi sind für einen großen Teil der denkbaren Anwendungen ausreichend, und Anwendungen können die Kommunikationsstrategie durch einen Wechsel zu einem anderen Modus gezielt ändern. Die genannten Modi wurden beispielhaft implementiert, um insbesondere den Modus-Wechsel zu evaluieren. Um neben dem Modus-Wechsel auch die Modi des Modus-Rahmenwerks direkt nutzen zu können, sollten diese durch bereits bestehende, auf die genannten Ziele spezialisierte Protokolle ersetzt werden. Auch eine Ausweitung der zur Verfügung stehenden Modi ist denkbar.

Eine zusätzliche Erweiterungsmöglichkeit betrifft den Modus-Wechsel an sich. Das Modus-Rahmenwerk geht davon aus, dass die Anwendung das Wissen über die momentane und zukünftige Anforderung an das Kommunikationsverhalten verfügt und deshalb den Anstoß zum Moduswechsel gibt. Führt man diesen Gedanken weiter, kann die Anwendung nicht nur einen vollständigen Wechsel auf einen anderen Modus initiieren, sondern kann auch die Ausführung eines Modus selbst beeinflussen. Bei Modus-Implementierungen, die selbst verschiedene Strategien anbieten, ist deshalb eine Erweiterung des Modus-Rahmenwerks denkbar: Anwendungen können dann auch das Kommunikationsverhalten innerhalb eines Modus beeinflussen. Insbesondere der direkte Eingriff auf Parameter der MAC-Ebene kann hier sowohl den Modus-Wechsel als auch die Modi selbst effizienter gestalten.

PanTalassa wurde erfolgreich auf einer Sensorknoten-Plattform von Crossbow implementiert und getestet. Aufgrund der ausschließlichen Verwendung genannter Platt-

form liegt ein Plattform-homogenes Sensornetz vor. Während die Flexibilität bei der Entwicklung, Ausführung und Ausbringung verteilter Anwendungen mit diesem Demonstrator gezeigt werden konnten, ist eine weitere Stärke des dienstorientierten PanTalassa-Ansatzes dabei nicht demonstrierbar: die Abstraktion von Betriebssystem und Plattform. Eine Portierung der virtuellen Maschine PanTalassaVM auf andere Sensorknoten-Betriebssysteme und andere Sensorknoten-Plattformen wäre deswegen ein weiterer Schritt, die Anwendungsmöglichkeiten bestehender Sensornetze auszuweiten, indem Sensornetze ohne Unterbrechung der laufenden Anwendung um zusätzliche Sensorknoten (auch „heterogener“ Art) erweitert werden können.

Im Zuge der Entwicklung von PanTalassa, insbesondere bei den Arbeiten zur Implementierung des intelligenten Gewächshauses, wurden auf Arbeitsebene mehrere einfache Editoren implementiert, die die Modellierung und Simulation von dienstorientierten Anwendungen ermöglichen. Die werkzeuggestützte Modellierung und Simulation förderte sowohl die Entwicklungsgeschwindigkeit als auch das Verständnis komplexer verteilter Anwendungen, wie der des intelligenten Gewächshauses. Allerdings schöpfen die entwickelten Werkzeuge bei weitem noch nicht das gesamte Potential aus. Die Entwicklung einer umfassenden Werkzeugpalette für die Definition von Diensten und Daten, graphische Notation, Simulation von Anwendungen, automatische Syntaxprüfung, und die Möglichkeit zur Ausbringung, Überwachung und Kontrolle von PanTalassa-Anwendungen würde die Nutzung realer Sensornetze erheblich vereinfachen. PanTalassa bietet hier ein großes Potential, um in Zukunft reale Sensornetze effizienter zu nutzen.

A. Simulationsparameter für Evaluation der Verfahren zur Verschiebung der Dienste

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsprotokoll für Schicht 3	AODV
Kommunikationsreichweite	250 m
Speicherkapazität der Knoten	unendlich
Mindestenergie e_{min} der Knoten	100
Energie der Knoten	220 - 740 (in Schritten von 20)
Energieverbrauch für Senden	$e_{send}(t) = 50 \cdot t$
Energieverbrauch für Empfangen	$e_{receive}(t) = 5 \cdot t$
Verfahren zur Verschiebung der Dienste	Zufall, Energie-bewusst, Optimiert Energie-bewusst
Metrik f bei Dienstverschiebung	$f = min(1000/(h+1); e; s)$
Metrik g für Dienstverschiebung	$g = (e < 100 \wedge s < \infty)$
Anzahl der Dienste	40
Anzahl der Dienstaufrufe pro Intervall	40 Dienste $\cdot$ 3 Aufrufe/Dienst = 120 Aufrufe
Intervall der Dienstaufrufe	15 Sekunden
Simulierte Zeit	5000 Sekunden ($\sim$ 1,4 Stunden)
Anzahl der Dienstaufrufe pro Simulation	120 Aufrufe $\cdot$ (5000/15) = 40000
Anzahl der Simulationen pro Stützpunkt	200

Tabelle A.1.: Simulationsparameter zur Untersuchung der Verfahren zur Verschiebung der Dienste

B. Simulationsparameter für Evaluation der Modus-Wechsel-Protokolle

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Zeroknowledge*
Cache-Größe	0 (*Zeroknowledge*)
Anzahl der Modi	3
Gültigkeitsdauer der Modi	5 Sekunden
Simulierte Zeit	25 Sekunden
Anzahl der Simulationen	1

Tabelle B.1.: Simulationsparameter zur Betrachtung der Verhaltensweise von Modus-Wechsel-Protokoll *Zeroknowledge*

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Multiknowledge*
Cache-Größe	1
Anzahl der Modi	3
Gültigkeitsdauer der Modi	5 Sekunden
Simulierte Zeit	25 Sekunden
Anzahl der Simulationen	1

Tabelle B.2.: Simulationsparameter zur Betrachtung der Verhaltensweise von Modus-Wechsel-Protokoll *Multiknowledge* Cache-Größe 1

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Multiknowledge*
Cache-Größe	2
Anzahl der Modi	3
Gültigkeitsdauer der Modi	5 Sekunden
Simulierte Zeit	25 Sekunden
Anzahl der Simulationen	1

Tabelle B.3.: Simulationsparameter zur Betrachtung der Verhaltensweise von Modus-Wechsel-Protokoll *Multiknowledge* Cache-Größe 2

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Zeroknowledge* und *Multiknowledge*
Cache-Größe	0 (*Zeroknowledge*); 1-2 (*Multiknowledge*)
Anzahl der Modi	3
Anzahl der Modus-Wechsel	10
Gültigkeitsdauer der Modi	5 Sekunden
Simulierte Zeit	20 Sekunden
Anzahl der Simulationen	60

Tabelle B.4.: Simulationsparameter für Vergleich von Effektivität und Effizienz von *Zeroknowledge* und *Multiknowledge* über den zeitlichen Verlauf

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Zeroknowledge* und *Multiknowledge*
Cache-Größe	0 (*Zeroknowledge*); 1-4 (*Multiknowledge*)
Anzahl der Modi	2 - 10
Anzahl der Modus-Wechsel	10
Gültigkeitsdauer der Modi	10 Sekunden
Simulierte Zeit	100 Sekunden
Anzahl der Simulationen pro Stützpunkt	600

Tabelle B.5.: Simulationsparameter für Vergleich von Effektivität und Effizienz von *Zeroknowledge* und *Multiknowledge* am Ende der Simulationszeit bei 10 Modus-Wechseln

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Wechsel-Protokoll	*Zeroknowledge* und *Multiknowledge*
Cache-Größe	0 (*Zeroknowledge*); 1-4 (*Multiknowledge*)
Anzahl der Modi	2 - 10
Anzahl der Modus-Wechsel	50
Gültigkeitsdauer der Modi	10 Sekunden
Simulierte Zeit	100 Sekunden
Anzahl der Simulationen pro Stützpunkt	600

Tabelle B.6.: Simulationsparameter für Vergleich von Effektivität und Effizienz von *Zeroknowledge* und *Multiknowledge* am Ende der Simulationszeit bei 50 Modus-Wechseln

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Implementierungen	energiesparend, schnell, robust
Kommunikationsprotokoll für Modus-Implementierungen	AODV
Ausfallwahrscheinlichkeit der Knoten	0% - 20%
Simulierte Zeit	100 Sekunden
Anzahl der Simulationen pro Stützpunkt	100

Tabelle B.7.: Simulationsparameter für Vergleich der Modus-Implementierungen

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Implementierungen	energiesparend, schnell
Kommunikationsprotokoll für Modus-Implementierungen	AODV
Simulierte Zeit	80 Sekunden
Anzahl der Simulationen	1

Tabelle B.8.: Simulationsparameter für Vergleich des Modus-Wechsels von schnell auf energiesparend

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Implementierungen	energiesparend, schnell
Kommunikationsprotokoll für Modus-Implementierungen	AODV
Simulierte Zeit	120 Sekunden
Anzahl der Simulationen	1

Tabelle B.9.: Simulationsparameter für Vergleich des Modus-Wechsels von energiesparend auf schnell

Simulationsparameter	Wert
Anzahl der Knoten	100
Größe des Areals	1000 m x 1000 m
Kommunikationsreichweite	150 m
Modus-Implementierungen	energiesparend, robust
Kommunikationsprotokoll für Modus-Implementierungen	AODV
Simulierte Zeit	100 Sekunden
Anzahl der Simulationen	1

Tabelle B.10.: Simulationsparameter für Vergleich des Modus-Wechsels von energiesparend auf robust

Literaturverzeichnis

[ABC+04] Abdelzaher, T., B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru und A. Wood: *EnviroTrack: towards an environmental computing paradigm for distributed sensor networks.* In: *Proceedings of the 24th International Conference on Distributed Computing Systems*, Seiten 582–589, 2004.

[Bal10] Balzert, Helmut: *Java: Objektorientiert programmieren.* W3L GmbH Witten Herdecke, 2. Auflage Auflage, 2010.

[BBD+02] Barro, R., J. Bicket, D. Dantas, B. Dur, T. Kim, B. Zhou und E. Sirer: *The Need for System-Level Support for Ad hoc and Sensor Networks.* 2002.

[BHS03] Boulis, Athanassios, Chih-Chih Han und Mani B. Srivastava: *Design and Implementation of a Framework for Efficient and Programmable Sensor Networks.* Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys '03), ACM Press, Seiten 187–200, Mai 2003.

[BHSS07] Boulis, A., C.-C. Han, R. Shea und M. B. Srivastava: *SensorWare: Programming sensor networks beyond code update and querying.* Elsevier Pervasive and Mobile Computing Journal, 3(4), 2007.

[BK10] Baumer, Susanne und Martin Krogmann: *Sensornetz-Baukasten: Passendes Werkzeug für jede Gelegenheit.* In: *VμE-Nachrichten*, Nummer 38, Seite 8. Fraunhofer-Verbund Mikroelektronik VμE, 2010.

[CJ03] Clausen, T. und P. Jacquet: *Optimized Link State Routing Protocol (OLSR).* IETF, RFC 3626, 2003.

[Cro] Crossbow Technology Inc.: *MicaZ Sensorknoten.* `http://www.xbow.com`, Datenblatt: `http://courses.ece.ubc.ca/494/files/MICAz_Datasheet.pdf` (letzter Zugriff: 20.12.2011).

[CWW10] Challen, Geoffrey Werner, Jason Waterman und Matt Welsh: *IDEA: Integrated Distributed Energy Awareness for Wireless Sensor Networks.* In: *Proceedings of 8th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '10)*, 2010.

[Dai11] Daimler AG: *Mercedes-Benz Deutschland - TechCenter - Klimatisierungsautomatik THERMOTRONIC*, 2011. `http://www.mercedes-benz.de/content/germany/mpc/mpc_germany_website/de/home_mpc/passengercars/home/world/webspecials/techcenter.chaptertherm otronic.html/thermotronic/details` (letzter Zugriff: 20.12.2011).

[DCF+09] Dardari, D., A. Conti, U. Ferner, A. Giorgetti und M. Win: *Ranging with ultrawide bandwidth signals in multipath environments.* In: *Proceedings of IEEE*, Band 97, 2009.

[Deu07] Deutschlandfunk: *Welt der Maschinen*, Oktober 2007. Sendung „Computer&Kommunikation" vom 3.10.2007, `http://www.dradio.de/dlf/sendungen/computer/680580/` (letzter Zugriff: 20.12.2011).

[Deu10] Deutschlandfunk: *Enganliegend und federleicht - Sensorhemd für Jogger und Patienten*, Februar 2010. Sendung „Forschung aktuell" vom 05.02.2010, `http://www.dradio.de/dlf/sendungen/forschak/1120027/` (letzter Zugriff: 20.12.2011).

[DH11] Darwish, Ashraf und Aboul Ella Hassanien: *Wearable and Implantable Wireless Sensor Network Solutions for Healthcare Monitoring.* Sensors, 11:5561–5595, 2011.

[DHK+09] Dudek, Denise, Christian Haas, Andreas Kuntz, Martina Zitterbart, Daniela Krüger, Peter Rothenpieler, Dennis Pfisterer und Stefan Fischer: *A Wireless Sensor Network For Border Surveillance (Demo).* In: *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, Berkeley, CA, USA, 2009.

[DOH07] Dunkels, Adam, Fredrik Osterlind und Zhitao He: *An Adaptive Communication Architecture for Wireless Sensor Networks.* In: *SenSys '07*, Sydney, Australia, November 2007.

[FR05] Frank, Christian und Kay Römer: *Algorithms for Generic role Assignement in Wireless Sensor Networks.* ACM International Conference on Embedded Networked Sensor Systems (SenSys), November 2005. San Diego, USA.

[Ful02] Fulford, Benjamin: *Sensors gone wild.* Forbes, 170(9):306, 28. Oktober 2002.

[GKGM05] Gummadi, Ramakrishna, Nupur Kothari, Ramesh Govindan und Todd Millstein: *Kairos: a macro-programming system for wireless sensor networks*. In: *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, New York, NY, USA, 2005. ACM.

[GMM+08] Gauger, Matthias, Daniel Minder, Pedro José Marrón, Arno Wacker und Andreas Lachenmann: *Prototyping sensor-actuator networks for home automation*. In: *Proceedings of the workshop on Real-world wireless sensor networks*, REALWSN '08, Seiten 56–60, New York, NY, USA, 2008. ACM.

[GZ06] Goos, Gerhard und Wolf Zimmermann: *Vorlesungen über Informatik, Band 2: Objektorientiertes Programmieren und Algorithmen*. Springer, Berlin, 4., überarbeitete Auflage, 2006.

[HB07] Hefeeda, M. und M. Bagheri: *Wireless Sensor Networks for Early Detection of Forest Fires*. In: *IEEE Internatonal Conference on Mobile Adhoc and Sensor Systems, 2007. MASS 2007.*, Seiten 1–6. IEEE, 2007.

[HCB00] Heinzelmann, W., A. Chandrakasan und H. Balakrishnan: *Energy-efficient communication protocol for wireless sensor networks*. Proceedings of 33rd Hawaii International conference System Sciences (HICSS'00), Januar 2000. Hawaii, USA.

[Hei10] Heise: *Handys helfen im Notfall beim Flüchten*, Mai 2010. `http://www.heise.de/newsticker/meldung/Handys-helfen-im-Notfall-beim-Fluechten-993792.html` (letzter Zugriff: 20.12.2011).

[Her07] Hergenröder, Anton: *Vom Modell zur Realisierung eines dienstorientierten Sensornetzes*. Diplomarbeit, Institut für Telematik an der Universität Karlsruhe (TH), Karlsruhe Institute of Technology (KIT), Karlsruhe, Deutschland, 2007.

[HHH+07] Hof, Hans-Joachim, Bernhard Hurler, Anton Hergenröder, Christian Haas und Michel Conrad: *Programming and Securing Service-oriented Wireless Sensor Networks*. In: *5th International Conference on Mobile Systems, Applications, and Services (MobiSys2007)*, 2007. Demo Abstract.

[HHZ04] Hurler, Bernhard, Hans-Joachim Hof und Martina Zitterbart: *A General Architecture for Wireless Sensor Networks: First Steps*. In: *4th International Workshop on Smart Appliances and Wearable Computing, Tokyo, Japan*, Seiten 442–444, 2004.

[HM06] Hadim, Salem und Nader Mohamed: *Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks.* IEEE DISTRIBUTED SYSTEMS ONLINE, IEEE Computer Society, 7(3):1541–4922, März 2006.

[Hof08] Hof, Hans-Joachim: *Sichere Dienste-Suche in Sensornetzen.* Dissertation am Institut für Telematik an der Universität Karlsruhe (TH), Karlsruhe Institute of Technology (KIT), Karlsruhe, Deutschland, 2008.

[HPW+05] Hurler, Bernhard, Christian Prehofer, Qing Wei, Martina Zitterbart und Joerg Claussen: *Method and apparatus for switching network modes.* Europäisches Patent (AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IS IT LI LT LU LV MC NL PL PT RO SE SI SK TR), Application No./Patent No. 05107985.3-, Applicant/Proprietor: DoCoMo Communications Laboratories Europe GmbH, 31.08. 2005.

[HSLA03] He, T., J. A. Stankovic, C. Lu und T. F. Abdelzaher: *SPEED: A Stateless Protocol for Real-Time communication in Sensor Networks.* International Conference on Distributed Computing Systemes (ICDCS), 2003.

[HZ04] Hurler, Bernhard und Martina Zitterbart: *A Flexible Concept to Program and Control Wireless Sensor Networks.* In: *Proceedings of the First European Workshop on Wireless Sensor Networks, Berlin, Germany*, 2004.

[ICT11] ICT Centre: *TasMAN: The Tasmanian Marine Analysis Network*, 2011. `http://research.ict.csiro.au/research/labs/tasictc/research-projects-1/tasman-the-tasmanian-marine-analysis-network` (letzter Zugriff: 20.12.2011).

[IEE11] IEEE: *IEEE-SA - Registration Authority*, 2011. `http://standards.ieee.org/develop/regauth/oui/public.html` (letzter Zugriff: 20.12.2011).

[IGE+03] Intanagonwiwat, C., R. Govindan, D. Estrin, J. Heidemann und F. Silva: *Directed Diffusion for Wireless Sensor Networking.* IEEE/ACM Transactions on Networking, 11:2–16, 2003.

[JHM07] Johnson, D., Y. Hu und D. Maltz: *The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4.* IETF, RFC 4728, 2007.

[JL10] Jindal, Apoorva und Mingyan Liu: *Networked Computing in Wireless Sensor Networks for Structural Health Monitoring.* In: *Electrical Engineering*, Band 7983, Seiten 798312–798313. SPIE, 2010.

[JOLR+03] Jovanov, E., A. O'Donnell Lords, D. Raskovic, P.G. Cox, R. Adhami und F. Andrasik: *Stress monitoring using a distributed wireless*

intelligent sensor system. In: *Engineering in Medicine and Biology Magazine*, Band 22, Seiten 49–55. IEEE, 2003.

[KGV+03] Kwon, Taek Jin, M. Gerla, V.K. Varma, M. Barton und T.R. Hsing: *Efficient flooding with passive clustering-an overhead-free selective forward mechanism for ad hoc/sensor networks.* In: *Proceedings of the IEEE*, Band 91, Seiten 1210–1220, 2003.

[KHB99] Kulik, J., W. R. Heinzelmann und H. Balakrishnan: *Adaptive protocols for information dissemination information in wireless sensor networks.* Proceedings of 5th ADM/IEEE Mobicom Conference (MobiCom'99), Seiten 174–185, August 1999. Seattle, USA.

[KPC+06] Kim, Sukun, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser und Martin Turon: *Wireless Sensor Networks for Structural Health Monitoring.* In: *Signal Processing*, Band 76, Seiten 1–22. ACM Press, 2006.

[KWA+03] Kumar, Rajnish, Mathew Wolenetz, Bikash Agarwalla, Jun-Suk Shin, Phillip Hutto, Arnab Paul und Umakishore Ramachandran: *DFuse: A Framework for distributed data fusion.* Proceedings of the 1st international conference on Embedded networked sensor systems, ACM Press, Seiten 114–125, 2003.

[KZ07] Kuntz, Andreas L. und Martina Zitterbart: *ServiceCast: Eine Architektur zur dienstorientierten Kommunikation in selbstorganisierenden Sensor-Aktor-Netzen.* In: *6. Fachgespräch Sensornetzwerke der GI/ITG Fachgruppe 'Kommunikation und Verteilte Systeme'*, Seiten 39–42, Juli 2007.

[LBJ03] Lipman, Justin, Paul Boustead und John Judge: *Neighbor aware adaptive power flooding (NAAP) in mobile ad hoc networks.* In: *International Journal of Foundations of Computer Science (IJFCS)*, Band 14, Seiten 237–252, 2003.

[LBY10] Li, Celia, Nirupama Barua und Cungang Yang: *Energy Efficient Role-based Clustering Algorithm for Wireless Sensor Networks.* Journal of Information and Communication Technology, 3(1), 2010.

[LC02] Levis, Philip und David Culler: *Maté: A Tiny Virtual Machine for Sensor Networks.* 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), Oktober 2002. ACM, San Jose, USA.

[LM03] Liu, Ting und Margaret Martonosi: *Impala: a middleware system for managing autonomic, parallel sensor systems.* In: *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, Seiten 107–118, New York, NY, USA, 2003. ACM.

[MA01] Manjeshwar, A. und D.P. Agrawal: *TEEN: a routing protocol for enhanced efficiency in wireless sensor networks.* Proceedings of the 15th International Parallel and Distributed Processing Symposium, Seiten 2009–2015, April 2001.

[MFHH03] Madden, Samuel, Michael J. Franklin, Joseph M. Hellerstein und Wei Hong: *The Design of an Acquisitional Query Processor for Sensor Networks.* ACM SIGMOD Conference, Seiten 491–502, 2003.

[MFHH05] Madden, Samuel, Michael J. Franklin, Joseph M. Hellerstein und Wei Hong: *TinyDB: An Acquisitional Query Processing System for Sensor Networks.* ACM Transactions on Database Systems, 30(1):122–173, 2005.

[MI05] Mahgoub, Imad und Mahammad Ilyas: *Handbook of Sensor Networks.* CRC Press, 2005.

[MOH04] Martinez, K., R. Ong und J.K. Hart: *Glacsweb: a sensor network for hostile environments.* In: *Proceedings of IEEE 1st International Conference on Sensors and Ad-hoc networks (SECON)*, Seiten 81–87. IEEE, 2004.

[MP11] Mottola, Luca und Gian Pietro Picco: *Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art.* ACM Computing Surveys, 43(3), April 2011.

[MR03] Mattern, Friedemann und Kay Römer: *Drahtlose Sensornetze.* Informatik Spektrum, 26(3):191–194, 2003.

[Nok08] Nokia: *Nokia Eco Sensor Concept*, 2008. `http://ncomprod.nokia.com/environment/devices-and-services/devices-and-accessories/future-concepts/eco-sensor-concept` (letzter Zugriff: 20.12.2011).

[ns2] *The Network Simulator - ns-2.* `http://www.isi.edu/nsnam/ns/` (letzter Zugriff: 20.12.2011).

[Oeh10] Oehler, Frank: *Drahtlose Kommunikation in Logistikanwendungen.* In: *VμE-Nachrichten*, Nummer 38, Seite 7. Fraunhofer-Verbund Mikroelektronik VμE, 2010.

[PBRD03] Perkins, C., E. Belding-Royer und S. Das: *Ad hoc On-Demand Distance Vector (AODV) Routing*. IETF, RFC 3561, 2003.

[PGL+10] Paradiso, Joseph A., Jonathan Gips, Mathew Laibowitz, Sajid Sadi, David Merrill, Ryan Aylward, Pattie Maes und Alex Pentland: *Identifying and facilitating social interaction with a wearable wireless sensor network*. Personal Ubiquitous Computing, 14:137–152, 2010.

[PH09] Perillo, M. und W. Heinzelman: *An Integrated Approach to Sensor Role Selection*. IEEE Transactions on Mobile Computing, 8(5):709–720, Mai 2009.

[PHWZ05] Prehofer, Christian, Bernhard Hurler, Qing Wei und Martina Zitterbart: *A Framework for Network Mode Control in Wireless Sensor Networks*. Technischer Bericht, Institut für Telematik, Universität Karlsruhe (TH), Dezember 2005. ISSN 1613-849X.

[RKM02] Römer, K., O. Kasten und F. Mattern: *Middleware Challenges for Wireless Sensor Networks*. ACM SIGMOBILE Mobile Computing and Communications, 2002.

[RKP+09] Rothenpieler, Peter, Daniela Krüger, Dennis Pfisterer, Stefan Fischer, Denise Dudek, Christian Haas, Andreas Kuntz und Martina Zitterbart: *FleGSens - secure area monitoring using wireless sensor networks*. In: *Proceedings of the International Conference on Sensor Networks, Information, and Ubiquituous Computing (ICSNIUC 2009)*, Singapore, 2009.

[RR06] Robertson, Suzanne und James Robertson: *Mastering the Requirements Process*, Band 2. Auflage. Addison Wesley, Harlow, 2006. ISBN 0-321-41949-9.

[Sch97] Schöning, Uwe: *Theoretische Informatik - kurzgefaßt*. Spektrum Akademischer Verlag, 3. Auflage, 1997.

[SMP+04] Szewczyk, Robert, Alan Mainwaring, Joseph Polastre, John Anderson und David Culler: *An analysis of a large scale habitat monitoring application*. In: *In Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Seiten 214–226, 2004. http://www.greatduckisland.net (ehemaliger Webauftritt, nicht mehr erreichbar).

[SO09] Spiegel-Online: *Sensor Chips - Am Puls der Welt*, Dec 2009. `http://www.spiegel.de/netzwelt/gadgets/0,1518,665279,00.html` (letzter Zugriff: 20.12.2011).

[SSJ01] Shen, C.-C., C. Srisathapornphat und C. Jaikaeo: *Sensor information networking architecture and applications.* IEEE Personal Communications, 8(4):52–59, August 2001.

[STD+10] Souza, Paulo A. Jr. De, Greg P. Timms, Andrew Davie, Ben Howell und Stephen Giugni: *Marine Monitoring using Fixed and Mobile Sensor Nodes.* In: *Oceans*, Seiten 1–4. IEEE, 2010.

[Sto04] Stojmenovic, I.: *Geocasting with Guaranteed Delivery in Sensor Networks.* IEEE Wireless Communications, Dezember 2004.

[Stu07] Stutz, Daniel: *Service Cast, ein neues Kommunikationsparadigma für Sensor-Aktor-Netze.* Diplomarbeit, Institut für Telematik, Universität Karlsruhe (TH), 2007.

[Tin] TinyOS: *Betriebssystem für Sensorknoten.* `http://www.tinyos.net` (letzter Zugriff: 20.12.2011).

[Tol05] Tolle, Christian: *Eine Beschreibungssprache für diensteorientierte drahtlose Sensornetze.* Diplomarbeit, Universität Karlsruhe (TH), 2005.

[TPT06] Tseng, Yu-Chee, Meng-Shiuan Pan und Yuen-Yung Tsai: *Wireless sensor networks for emergency navigation.* In: *Computer*, Band 39, Seiten 55–62. IEEE, 2006.

[VP05] Ververidis, C. N. und G. C. Polyzos: *Routing layer support for service discovery in mobile ad hoc networks.* In: *Third IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOMW'05)*, Seiten 258–262. Kauai Island, Hawaii, USA: IEEE Computer Society, März 2005. ISBN 0-7695-2300-5.

[WM04] Welsh, Matt und Geoff Mainland: *Programming sensor networks using abstract regions.* In: *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, Berkeley, CA, USA, 2004. USENIX Association.

[XRC+04] Xu, N., S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan und D. Estrin: *A Wireless Sensor Network for Structural Monitoring.* In: *In Proceedings of the ACM Conference on Embedded Networked Sensor Systems(Sensys04)*, 2004.

[YAEW04] Younis, Mohamed, Kemal Akkaya, Mohamed Eltoweissy und Ashraf Wadaa: *On Handling QoS Traffic in Wireless Sensor Networks.* Proceedings of the 37the Annual Hawaii International conference on System Sciences (HICSS'04), 2004. Hawaii.

[YCLZ01] Ye, F., A. Chen, S. Liu und L. Zhang: *A scalable solution to minimum cost forwarding in large sensor networks.* Proceedings of 10th International Conference Computer Communication Networks (ICCCN), Seiten 304–309, 2001.

[YG02] Yao, Yong und Johannes Gehrke: *The cougar approach to in-network query processing in sensor networks.* SIGMOD Rec., 31:9–18, September 2002.

[YGT03] Yungjung, Yi, M. Gerla und Jin Kwon Taek: *Efficient flooding in ad hoc networks: a comparative performance study.* In: *ICC '03. IEEE International Conference on Communications*, Band 2, Seiten 1059–1063, 2003.

[Zig11] ZigBee Alliance: *ZigBee (IEEE 802.15.4): Standard für drahtlose Kommunikation*, 2011. `http://www.zigbee.org` (letzter Zugriff 20.12.2011).