Institut für Parallele und Verteilte Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 1960

Energieeffiziente Dienstnutzung in spontan vernetzten ubiquitären Rechnersystemen

Ralph Lange

Studiengang:	Informatik
Prüfer:	Prof. Dr. Kurt Rothermel
Betreuer:	Gregor Schiele
begonnen am:	15. Juli 2004
beendet am:	14. Januar 2005
CR-Klassifikation:	C.2.2, C.2.4, D.4.4

Abstract

In ubiquitous computing – the third era of the electronic data processing – energy efficient algorithms play an important role, since the corresponding computer systems have limited energy supplies. Many tasks are mastered through cooperation in ubiquitous computer systems. The computers communicate via wireless networks with each other. The network interface of such a computer consumes a lot of energy. Energy can be saved by deactivating this interface, but then the computer becomes unreachable for others.

Today there are a lot of energy efficient algorithms for media access control and routing. These algorithms save energy by deactivating the network interface during regular intervals and try to compensate the unreachability.

In this work energy efficient algorithms, which use knowledge on the applications and services of a computer, are examined. Using this knowledge potentially most of all energy can be saved. At the same time negative effects of the deactivation of the network interfaces can be minimized. Here, especially variants or extensions of SANDMAN, an algorithm for energy efficient service discovery, are examined. SANDMAN divides a wireless network in clusters and uses a clusterhead per cluster as service directory. In the approach discussed here, not only the service discovery but also the use of the service occurs via the clusterhead. The real service is hidden behind the clusterhead.

In the first part of this work advantages and disadvantages of this approach are examined. One result among others is how services and computers have to look like to be suitable for this approach. After the discussion of some existing energy efficient algorithms, LATE RISER, a concrete algorithm according to the described approach, is developed and its implementation is described. Data of the analysis of LATE RISER through simulation are presented and discussed.

The conclusion is that using the described approach a lot of energy can be saved, but it can only be used under suitable circumstances.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Miniaturisierung und Vernetzung prägen die Vision des Ubiquitous Computing. Alltagsdinge werden in „smarte“ Gegenstände verwandelt, um in Form ubiquitärer Rechnersysteme den Menschen in unaufdringlicher und auf intuitive Weise zu unterstützen [MatLan 2003]. Werden Rechner heute als solche explizit wahrgenommen, wie PDAs,[1] Laptops oder in Handys, so sollen die Rechner der Zukunft unsichtbar im Hintergrund agieren. Von der Glühbirne, die sich dimmt, sobald der Letzte den Raum verlässt, bis zum Rasensprenger, der im Internet selbstständig nach der aktuellen Wettervorhersage sucht, reichen die Ideen der Entwickler und Erfinder.

Prozessoren, das Herzstück eines jeden elektronischen Rechners, sind in den vergangenen Jahrzehnten nicht nur immer schneller, sondern auch erheblich billiger geworden und können heute mit einer Vielzahl zusätzlicher Funktionen ausgestattet werden. Über winzige Sensoren können solche kleinsten Rechner ihre Umgebung erfassen und mit speziellen Schnittstellen drahtlos kommunizieren. Werden solche Rechner in andere Objekte integriert, so spricht man von „Smart Devices“. Wegen ihrer Spezialisierung sind die Rechner in Smart Devices nicht mit den Universalrechnern des 20. Jahrhunderts vergleichbar.

Smart Devices verfügen über einen sehr begrenzten Energievorrat, da sie ihre Energie oft aus einem Akkumulator oder einer Batterie beziehen. Die verschiedenen Komponenten der Miniaturrechner müssen daher darauf ausgelegt sein, so energieeffizient wie möglich zu arbeiten.

Neben den offensichtlichen Anforderungen an die Hardware der ubiquitären Rechnersysteme werden auch an die Software eine Reihe von neuen Anforderungen gestellt. Die Software muss garantieren, dass Smart Devices autonom arbeiten können. Eingriffe durch den Benutzer sind in der Regel nicht möglich. Das gilt für die Anpassung an die Umgebung ebenso wie für die Behandlung von Anwendungsfehlern.

Die meisten Aufgaben werden im Ubiquitous Computing durch Kooperation bewältigt. Im oben genannten Szenario des intelligenten Rasensprengers kann dieser vielleicht über einen lokalen Sensor die Feuchte des Bodens ermitteln, doch er wird über keinen Internetzugang verfügen, um die genannten Wetterdaten abzurufen. Dies wird erst in Kooperation mit einer Basisstation oder über ein Mobiltelefon möglich.

Der Rasensprenger muss daher selbstständig einen geeigneten Internetzugangsdienst finden. Das Suchen und Auffinden solcher Dienste stellt in einem zunächst völlig unbekannten Netz keine triviale Aufgabe dar. Man nennt das Suchen und automatische Finden fremder Dienste kurz *Diensterkennung*, nach dem Englischen auch *Service Discovery*. Diensterkennung ist ein gut erkundetes Forschungsgebiet. Heute existieren viele kommerziell eingesetzte Verfahren.

Die meisten Verfahren setzen fest installierte Netze voraus, wie sie in Bürogebäuden oder Hotels anzutreffen sind. Oft wird Diensterkennung und Kooperation in infrastrukturlosen Systemen, so genannten *Ad-hoc-Netzen*, benötigt. Da es vorab keinen ausgezeichneten Rechner gibt, an den sich die anderen Rechner zur Diensterkennung wenden können, stellt die Diensterkennung dort eine besondere Herausforderung dar.

[1] Personal Digital Assistant

Doch auch für Ad-hoc-Netze gibt es inzwischen eine Reihe von Verfahren, die Diensterkennung ermöglichen. Allerdings wurde bei den meisten Entwicklungen der oft begrenzte Energievorrat der Rechner nicht berücksichtigt. Die Verfahren gehen mit der Ressource Energie verschwenderisch um, weil sie unnötig viel kommunizieren und Daten mehrfach oder ohne Bedarf übertragen.

Nur zwei Verfahren – DEAPSpace [Nidd 2001] und SANDMAN [ScBeRo 2004] – adressieren Energieeffizienz bei der Diensterkennung. SANDMAN ist an der Universität Stuttgart entstanden und ermöglicht bisher nur die Erkennung von Diensten. In dieser Arbeit wird untersucht, ob auch die energieeffiziente *Nutzung* von Diensten mit einer Variante oder Erweiterung von SANDMAN möglich ist.

1.1 Aufgabe

Das energieeffiziente Diensterkennungverfahren SANDMAN ist aus den beiden Protokollen SEDUCE-1 und -2 [Angstm 2003] entstanden. Wie der Name „Service Awareness and Discovery for Mobile Ad hoc Networks“ ausdrückt, ist SANDMAN vor allem für spontan vernetzte Systeme geeignet. Da ein ausgezeichneter Knoten[2] zunächst fehlt, müssen sich die Knoten in einem solchen Netz gemeinsam mit Hilfe verteilter Algorithmen organisieren.

SANDMAN bildet dazu logische Gruppen, so genannte Cluster, und wählt je Cluster einen Clusterhead. Dieser arbeitet als Dienstverzeichnis, das heißt, er speichert die Dienstbeschreibungen aller Knoten im Cluster und beantwortet Diensterkennungsanfragen. Dadurch können die anderen Knoten des Clusters regelmäßig in einen energiesparenden Schlafmodus wechseln.

Dieses Prinzip soll nun von der Diensterkennung auf die Dienstnutzung übertragen werden. So wie bisher die Diensterkennungsanfragen vom Clusterhead, statt von dem Knoten, der den passenden Dienst bietet, beantwortet werden, soll nun auch die Dienstnutzung über den Clusterhead ablaufen. Da der Clusterhead den Dienst natürlich nicht selber bieten kann, leitet er die Anfragen an einen geeigneten Knoten weiter.

1.2 Aufbau dieser Arbeit

Im folgenden Kapitel wird eine Einführung in Ubiquitous Computing und ubiquitäre Rechnersysteme gegeben und weitere Fachbegriffe erklärt und definiert.

In Kapitel 3 werden zunächst der Energieverbrauch in drahtlosen Netzen und der energieeffiziente Algorithmus SANDMAN im Detail erläutert. Anschließend werden allgemeine Anforderungen an ein Verfahren zur energieeffizienten Dienstnutzung und spezielle Anforderungen in dieser Arbeit genannt. Mit diesem Wissen werden in Kapitel 4 Vor- und Nachteile eines solchen Verfahrens untersucht und bewertet.

In Kapitel 5 wird ein Einblick in existierende Arbeiten zur Energieeffizienz in Ad-hoc-Netzen gegeben. Da das Thema energieeffiziente Dienstnutzung in der hier betrachteten Weise bisher kaum bearbeitet wurde, werden auch Arbeiten zu Media Access Control und Routing, die für diese Arbeit relevant sind, betrachtet.

[2] So werden bei der Beschreibung von Netzen die verschiedenen Geräte und Rechner genannt.

Daraufhin wird in Kapitel 6 ein konkretes Verfahren entwickelt und dessen Implementierung in Kapitel 7 erläutert. Ein weiteres Kapitel ist der Analyse von LATE RISER durch Simulation gewidmet.

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 9.

2 Begriffserklärungen und Definitionen

In diesem Kapitel werden häufig verwendete Begriffe eingeführt und Einblicke in Ubiquitous Computing und Ad-hoc-Netze gegeben.

2.1 Ubiquitous Computing

Der Begriff „Ubiquitous Computing“, wörtlich „Allgegenwärtiges Rechnen“, geht auf Mark Weiser, einem Forscher am Xerox Palo Alto Research Center (PARC), zurück. Im dortigen Computer Science Laboratory wurde bereits 1988 ein Forschungsbereich Ubiquitous Computing eröffnet. Mark Weiser legte mit seinem Artikel „The Computer for the 21st Century“ [Weiser 1991] die Grundlagen für das, was heute unter Ubiquitous Computing verstanden wird.

Mark Weiser nennt jene Technologien besonders ausgereift, die nicht mehr explizit als solche wahrgenommen werden. Er führt Schrift, zur symbolischen Repräsentation von Sprache, als besonders gelungenes Beispiel an. Ubiquitous Computing soll Rechner in diesem Sinne „unsichtbar“ machen.

Ubiquitous Computing ist die dritte Ära der elektronischen Datenverarbeitung. Die Rechner der ersten Ära wurden von vielen Anwendern gleichzeitig genutzt, da Computer sehr teure Geräte waren. Die zweite Ära ist das Personal Computing. Jeder Anwender verfügt über einen eigenen Rechner. Doch die Rechner sind unhandlich und nach Mark Weiser beschränkt sich die „obskure Aura“, die das Personal Computing umgibt – jeder verwendet PCs, kaum einer versteht sie –, nicht nur auf die oft viel zu komplizierte Benutzerschnittstelle, sondern auf die Idee der universellen Rechner an sich. Das soll sich mit der dritten Ära ändern. Die physische Welt wird von vielen Rechnern durchdrungen sein, die die Menschen intuitiv unterstützen, aber nicht mehr als Rechner wahrgenommen werden. In [Kirste 2002, Kap. 1] werden fünf Merkmale ubiquitärer Rechnersysteme genannt:

1. *Einbettung:* Die Rechner der Zukunft werden nicht mehr die physische Welt nachbilden, wie zum Beispiel der Desktop in modernen Betriebssystemen, sondern die physische Welt mit Informationen anreichern.

   Mark Weiser und seine Kollegen haben „Pads“ entwickelt, nach eigenen Angaben eine Kreuzung zwischen einem Laptop und einem Blatt Papier. Diese Pads sollen auch wirklich wie Papier behandelt werden. Sie werden nicht wie ein Laptop überall mithin genommen, sondern wie ein Notizzettel nur bei Bedarf, da der Anwender für jede Aufgabe ein anderes Pad verwendet. Pads können wie Ordner oder Mappen auf dem Schreibtisch oder im Regal organisiert werden.

Die meisten Rechner werden bei Ubiquitous Computing in andere Geräte, zum Beispiel im Haushalt, eingebettet sein.

2. *Vernetzung:* Es werden nicht mehr einige wenige Geräte, sondern alle Gegenstände miteinander vernetzt. Die Vernetzung erfolgt spontan und ist dynamisch.

3. *Allgegenwart:* Ubiquitous Computing soll überall und zu jeder Zeit den Zugriff auf Informationen und Dienste ermöglichen. Verschiedene Dienste und Netze werden zu einem großen System verschmolzen.

4. *Kontext:* Bei Ubiquitous Computing haben die Rechner einen Bezug zur realen Welt. Sie verbinden die physische Welt mit der Welt der Informationsverarbeitung.

5. *Neue Geräte:* Ubiquitous Computing verwendet verschiedenste Geräte. Die Rechner der Zukunft unterscheiden sich untereinander stark in Größe, Rechenleistung und Ausstattung.

Synonym zu Ubiquitous Computing wird oft der Begriff „Pervasive Computing“ verwendet. Andere Autoren sehen einen kleinen Unterschied zwischen beiden Begriffen.

In [Mattern 2003] wird unter Pervasive Computing eine pragmatische, kommerzielle Form des Ubiquitous Computing verstanden, die mit Electronic-Commerce und webbasierten Geschäftsprozessen in der Praxis bereits Fuß gefasst hat.

2.2 Smart Devices

Ubiquitous Computing verwandelt Alltagsgegenstände durch die Verschmelzung mit Rechnern in Smart Devices. Nach [Kirste 2002, Kap. 1] haben Smart Devices Schnittstellen zu drei Bereichen:

1. *Zur realen Welt:* Mit kleinen Sensoren erfassen Smart Devices die physische Umgebung, in der sie sich befinden.

2. *Zum Anwender:* Die Benutzerinteraktion kann herkömmlich mit Bildschirm und Tastatur oder mit modernen Techniken wie Sprachein- und -ausgabe gestaltet sein. In den meisten Fällen wird sie allerdings nicht mehr explizit, sondern intuitiv und situationsbezogen erfolgen.

3. *Zur „digitalen Welt“:* Mit Funknetzkarten, Infrarotschnittstellen oder herkömmlich per Kabel kommunizieren Smart Devices mit anderen Geräten und erhalten Zugang zu größeren Systemen.

Eine Klassifizierung der Smart Devices ist schwierig. In [Weiser 1991] werden sie nach Größe, in Yard-, Foot- und Inch-size, unterteilt. In [Kirste 2002, Kap. 4] werden Smart Devices nach zwei Dimensionen, von *eingebettet* bis *mobil* und von *zur Kontrolle* bis *zur Kommunikation oder Information*, klassifiziert.

Die Betriebssoftware von Smart Devices unterscheidet sich deutlich von den Betriebssystemen bei Personal Computern. Sie muss in vielen Fällen auf sehr ressourcenbeschränkten

Geräten arbeiten und kann daher nicht den Umfang heutiger Betriebssysteme haben. Sie verwaltet weniger Betriebsmittel – viele Smart Devices verfügen über keinen externen Speicher – und muss oft Realzeit-Anforderungen genügen [Kirste 2002, Kap. 4]. Die Micro-Edition der Java 2 Platform [J2ME], ist ein bekanntes Beispiel für eine solche Betriebssoftware. Durch Profile kann sie an verschiedene Hardwareplattformen in Umfang und Funktionalität angepasst werden.

Da Ubiquitous Computing noch in den Kinderschuhen steckt, lassen sich heute weder bei der Hard- noch bei der Software alle Entwicklungen absehen. Exemplarisch werden nun vier bereits existierende Geräteklassen und deren Merkmale, insbesondere zum energieeffizienten Betrieb, genannt.

Personal Devices Zu den Personal Devices zählen Endgeräte mit ausführlicher Benutzerschnittstelle wie zum Beispiel Organizer, PDAs oder Mobiltelefone. Diese Geräte verfügen über eine relativ hohe Speicherkapazität und große Rechenleistung. Zur Kommunikation mit anderen Rechnern besitzen sie oft mehrere Schnittstellen, wie drahtloser Funk nach IEEE 802.11 oder Bluetooth, Funkmodems nach dem GSM, GPRS oder gar UMTS-Standard, Netzwerkanschlüsse für kabelgebundene IP-Netze und Infrarot-Schnittstellen.

Der Energievorrat solcher Geräte ist zwar relativ groß, doch die vielen Komponenten, zum Beispiel ein oft hintergrundbeleuchteter Bildschirm, haben auch entsprechend hohe Leistungsaufnahmen. Energiesparmaßnahmen sind bei einem solchen Gerät vor allem dann möglich, wenn der Benutzer es gerade nicht verwendet. Hierbei können auch etablierte Methoden wie das Abschalten von Festplatten, Prozessoren und Bildschirmen verwendet werden.

Sensoren Sensoren sind winzige Rechner ohne Benutzerschnittstelle. Sie haben eine geringe Speicherkapazität und Rechenleistung und können nur in Kooperation mit anderen Geräten sinnvoll arbeiten. Es gibt einen eigenen Forschungsbereich Sensornetze.

Sensoren zeichnen sich oft durch einen besonders kleinen Energievorrat aus und werden wegen ihrer geringen Herstellungskosten in großen Mengen produziert und eingesetzt. Einsatzgebiete sind die Erfassung von Umweltdaten in ökologischen Anwendungen oder in Katastrophengebieten und die Überwachung technischer Systeme. In solchen Szenarien ist energieeffiziente Kommunikation besonders wichtig, da die Energievorräte in der Regel nicht mehr aufgefrischt werden können.

Server und Basisstationen In die Klasse gehören zum Beispiel Print-, Mail-, File-Server und Access-Points. Sie verfügen über eine große Rechenleistung, können viele andere Geräte gleichzeitig bedienen und bieten oft Zugang zu größeren Systemen, wie dem Internet.

Da solche Geräte über das allgemeine Stromnetz und nicht aus einer Batterie versorgt werden, ist energieeffizientes Verhalten weniger wichtig.

Peripheriegeräte Zu dieser Klasse gehören einfache Geräte wie Funk-Mäuse oder -Tastaturen, aber auch größere Geräte wie Scanner, Drucker oder Anzeigetafeln. Peripheriegeräte treten in der Regel als Dienstgeber und nicht als Klienten auf.

2.3 Drahtlose Netze

Im Personal Computing erfolgte drahtlose Kommunikation bis vor einigen Jahren fast ausschließlich mit Infrarotschnittstellen [IrDA 1996]. Inzwischen wurden diese von Funknetzwerkkarten nach dem IEEE-Standard 802.11b [IEEE 802.11] abgelöst. Funknetze sind heute weit verbreitet. Zwei Eigenschaften drahtloser Netze sind in dieser Arbeit besonders wichtig:

1. Netzwerkschnittstellen haben in drahtlosen Netzen eine begrenzte Reichweite. Können sich zwei Knoten trotzdem direkt erreichen, so spricht man von einer *Single-Hop*-Verbindung.

   In vielen Fällen können zwei Knoten aber nur indirekt miteinander kommunizieren und benötigen andere Knoten – so genannte *Router* –, die Nachrichten zwischen ihnen weiterleiten. Man spricht von einer *Multi-Hop*-Verbindung.

2. In drahtlosen Netzen kann jeder Knoten eine Nachricht zwischen zwei anderen Knoten mithören, wenn er sich in Reichweite des Senders befindet.

   Jeder Knoten kann aber auch eine Nachricht zwischen zwei anderen Knoten stören, befindet er sich in Reichweite des Empfängers und sendet selber eine Nachricht. Man spricht von einer *Kollision* der Nachrichten. Kollisionen sind in der Regel aber keine böse Absicht, sondern eine logische Folge dessen, dass alle Knoten gleichzeitig dasselbe Übertragungsmedium – elektromagnetische Wellen desselben Frequenzbereichs – verwenden.

   Die Vermeidung von Kollisionen ist Aufgabe des *Media Access Control*.

Je nach Position befindet sich ein Knoten eines ubiquitären Rechnersystems in einem völlig anderen Kontext. Sogar wenn der Rechner seine Position nicht ändert, kann sich die Umgebung aus seiner Sicht laufend ändern. An diese Änderungen muss er sich entsprechend anpassen. Dabei lassen sich die möglichen Umgebungen und die damit verbundenen Betriebsarten in drei Klassen [Angstm 2003] aufteilen:

Isolierter Betrieb Im isolierten Betrieb ist ein Rechner völlig auf sich alleine gestellt. Personal Devices können dabei noch sinnvoll arbeiten. Sensoren hingegen können in einer solchen Umgebung den Dienst einstellen, sofern sie nicht über eine hohe Speicherkapazität verfügen und ihre Daten später abliefern können. Server und Basisstationen können im isolierten Betrieb in einen Energiesparmodus wechseln und warten bis wieder mobile Rechner in ihre Reichweite kommen und ihre Dienste benötigen. Ähnliches gilt für Peripheriegeräte.

Netze mit Infrastruktur In einem solchen Netz existiert eine Basisstation, die das Netz koordiniert. Zwischen ihr und den mobilen Geräten herrscht eine klare Aufgabenteilung. Die Basisstation bietet den mobilen Knoten nicht nur Zugang zu anderen Netzen oder entfernten Diensten, sondern kann auch die energieeffiziente Kommunikation der mobilen Knoten untereinander durch Zwischenpuffern von Nachrichten unterstützen.

In größeren Netzen können mobile Knoten „ihre“ Basisstation wechseln. Für einen reibungslosen Wechsel verwenden die Basisstationen untereinander ein entsprechendes Übergabeprotokoll.

Ad-hoc-Netze und MANETs Ad-hoc-Netze werden spontan und nur vorübergehend gebildet. Darin sind sich die meisten Definitionen, zum Beispiel [TechTa 2003, PatNag 2001], einig.

Dies beinhaltet auch, dass sich Ad-hoc-Netze selbstständig und ohne zentrale Koordinierung organisieren – es sind weder Benutzereingriffe noch vorab ausgezeichnete Knoten erforderlich. Daher ist die Aufgabenverteilung unter den Knoten zunächst unklar. Sie müssen sich gemeinsam darauf einigen. Man spricht auch von Peer-to-Peer-Netzen, um die Gleichrangigkeit aller Knoten auszudrücken.

Viele Definitionen verstehen unter Ad-hoc-Netzen sogar ausschließlich drahtlose, infrastrukturlose Netze. Nach dem IEEE Standard 802.11 für Funknetze [IEEE 802.11, Kap. 3] wird auch der Name Independent Basic Service Set (IBSS) verwendet.

In der Regel sind die Knoten eines Ad-hoc-Netzes mobil. In den in dieser Arbeit betrachteten Szenarien sind aber auch fest installierte Knoten denkbar, sofern sie das Netz nicht im Sinne einer Basisstation koordinieren.

Typisches Kennzeichen von Ad-hoc-Netzen ist eine hohe Dynamik. Knoten kommen und gehen nach Belieben und ändern damit auch ständig die Struktur des Netzes. In diesem Zusammenhang wird auch der Begriff Mobile Ad hoc Network (MANET) verwendet. Er bezeichnet im Besonderen Ad-hoc-Netze, die dank eines Routing-Protokolls – ein solches wird in IEEE 802.11 nicht vorgeschrieben – in ihrer Ausdehnung nicht auf die Sende- und Empfangsreichweite eines Knotens beschränkt sind.

Da Routing bei den hier betrachteten Netzen nicht zwingend vorgeschrieben ist, wird im Folgenden der Begriff Ad-hoc-Netz verwendet.

2.4 Cluster

Um mit Änderungen innerhalb des Netzes schnell und effizient umgehen zu können, werden für die Algorithmen in Ad-hoc-Netzen dynamische Kontrollstrukturen benötigt. Eine gängige Methode dafür ist die Clusterbildung.

Dazu werden die Knoten eines Netzes in logischen Gruppen, den Cluster, zusammengefasst. Innerhalb eines Clusters übernehmen ein oder mehrere Knoten, *Koordinatoren* oder *Clusterheads* genannt, die benötigten Kontrollaufgaben. Ein solcher Cluster ist mit seinem Clusterhead, kurz CH, in Abbildung 1 dargestellt.

Die Clusterbildung kann hierarchisch erfolgen [Perkins 2000, Kap. 4].

Viele hier betrachtete Algorithmen setzen ein geeignetes Clustermanagement voraus. Die Aufgaben des Clustermanagements sind Suche nach und Beitritt zu vorhandenen Clustern oder gegebenenfalls die Bildung eines neuen Clusters. Außerdem beteiligt es den Knoten an der Wahl eines neuen Clusterheads.

Das Clustermanagement sollte eine Schnittstelle bieten, über die Anwendungen Merkmale des Clusters erfragen können und bei wichtigen Änderungen benachrichtigt werden.

2.5 Dienste

Der Begriff Dienst wird im Bereich Software sehr vielseitig verwendet. Man kann darunter die verschiedensten Benutzerprogramme, Dienstprogramme für die Wartung bestimmter Hard-

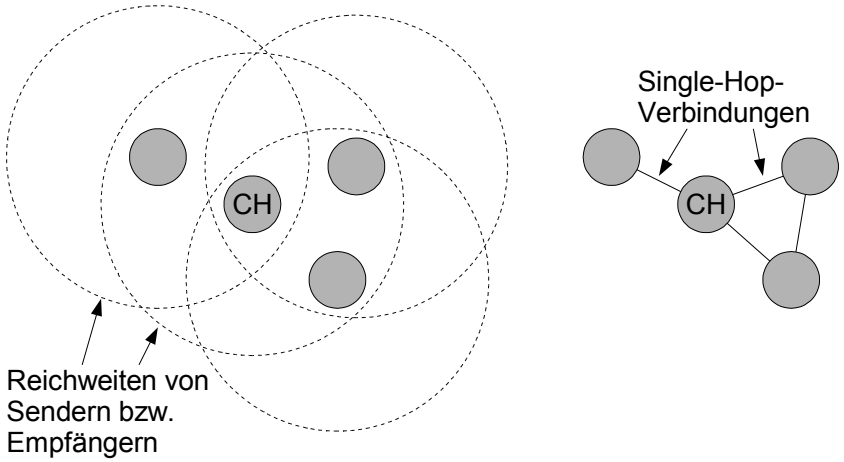

Abbildung 1: Cluster mit Clusterhead in zwei Darstellungen

warekomponenten oder Systemdienste, so genannte Daemons, verstehen. Auch eine virtuelle Maschine, wie die Java-VM, oder einzelne Objekte einer Anwendung werden manchmal als Dienst bezeichnet.

Entsprechend schwierig ist es den Begriff „Dienst“ zu definieren. Eine sehr allgemeine Definition findet sich in [Irmscher 2004, Kap. 5.1]:

> Ein Dienst ist eine Software-Instanz, die auf einem oder mehreren Rechnern ausgeführt wird.

Ubiquitous Computing In ubiquitären Rechnersystemen kann jeder Knoten eine Reihe von Software-Instanzen ausführen, die andere Knoten nutzen können. Jeder Knoten kann also Dienste anbieten und nutzen.

Dienste sind durch wohldefinierte Schnittstellen gekennzeichnet. Diese sind so beschaffen, dass die Nutzung ohne manuelle Anpassung und über das Netz – in einem *entfernten Aufruf* – erfolgen kann. Die Semantik eines Dienstes kann dabei implizit oder explizit mit der Schnittstelle beschrieben werden. Dienste können für die Erbringung ihrer Leistung auf andere Dienste zurückgreifen. So kann sich ein Dienst aus verschiedenen Teildiensten zusammensetzen.

Klassifikation Dienste in ubiquitären Rechnersystemen können nach fünf Dimensionen charakterisiert werden [KoObKl 2004, AP. 1.2]:

1. *Zusammensetzung:* Ein Dienst kann für die Diensterbringung keine, statisch bekannte, dynamisch bekannte oder dynamisch unbekannte Teildienste auf entfernten Knoten benötigen.

2. *Kooperationsform:* Man unterscheidet zwischem dem Client-Server-Modell und Mehrparteiendiensten. Im Client-Server-Modell interagieren genau zwei Partner in eindeu-

tigen Rollen. In Mehrparteiendiensten können beliebig viele Knoten ohne eindeutige Rollen für eine Diensterbringung kooperieren.

3. *Zeitkopplung:* Es werden synchrone Dienste und asynchrone Dienste unterschieden. Bei den synchronen Diensten wartet der Klient nach dem Senden der Anfrage, bis die Antwort eintrifft. Bei asynchronen Diensten wartet der Klient nicht, sondern wird später beim Eintreffen der Antwort über deren Vorliegen informiert.

4. *Ergebnisproduktion:* Der Dienst kann das Ergebnis schrittweise in Teilergebnissen oder als Ganzes liefern.

5. *Benutzung:* Dienste können automatisch oder interaktiv sein. Bei Letzteren sind für die Diensterbringung Benutzereingriffe erforderlich.

Diese Liste erhebt keinen Anspruch auf Vollständigkeit. Es lassen sich viele weitere Merkmale finden, nach denen Dienste klassifiziert werden können. Wichtige weitere Merkmale sind zum Beispiel die benötigte Rechenleistung und die Verwendung von Kontextwissen über Klienten.

3 Motivation

Energieeffizienz in Ad-hoc-Netzen ist ein sehr junges Forschungsgebiet. Zwar existieren inzwischen einige entsprechende Algorithmen, doch beschränken sich diese meistens auf die reine Nachrichtenübertragung oder spezielle Dienste.

Erkenntnisse und Verfahren zur energieeffizienten Nutzung beliebiger Dienste *mit* Verwendung von Anwendungswissen sind spärlich. Diese Arbeit soll einen Schritt zur Schließung dieser Lücke beitragen.

3.1 Energie

Energie kann bei Rechnern nur durch Abschalten oder Herunterdrosseln von Komponenten gespart werden. Eine Komponente, die besonders bei kleinen Rechnern relativ viel Energie verbraucht, ist die drahtlose Netzwerkschnittstelle, sehr häufig in Form einer Funknetzkarte.

Die drahtlose Netzwerkschnittstelle kann durch Abschalten des Verstärkers oder der ganzen Komponente schnell in einen Energiesparmodus versetzt werden. In diesem Modus kann der Knoten aber weder senden noch empfangen, noch durch ein Signal geweckt werden. Genau darin liegt die eigentliche Herausforderung!

Dieser Modus wird fortan *Schlafmodus* genannt. Ein Knoten *schläft*, wenn er keine Nachrichten senden oder empfangen kann und damit nichts über den aktuellen Zustand des Netzes erfährt. Andernfalls ist der Knoten *wach*.

Während des Schlafmodus' können auch andere Komponenten eines Rechners in einen energiesparenden Zustand wechseln. Beispiele dafür sind das Reduzieren von Chipfrequenzen oder das Abschalten von Festplatten. Solche Maßnahmen haben aber keinen Einfluss auf die Erreichbarkeit eines Rechners im Netz und können daher ohne weitere Vorkehrungen verwendet werden. Sie werden hier deshalb nicht explizit betrachtet.

3.2 SANDMAN im Detail

Das energieeffiziente Diensterkennungsverfahren SANDMAN [ScBeRo 2004] geht davon aus, dass jeder Knoten eines Ad-hoc-Netzes Dienste anbieten und nutzen kann. Daher werden alle Knoten gleich behandelt.

Die Knoten des Netzes werden in Cluster organisiert. Zur Bildung der Cluster und deren Verwaltung greift SANDMAN auf existierende Algorithmen zurück. Je Cluster wird genau ein Clusterhead gewählt. Clusterhierarchien kommen nicht zum Einsatz. Ein Clusterhead muss jeden Knoten seines Clusters in einer Single-Hop-Verbindung erreichen können. Verfahren zur Bildung von Clustern, die dieser Anforderung genügen, lassen sich bei vielen Routing-Algorithmen finden [SriChi 2002, YaBi+ 2003]. Wird daher im Netz clusterbasiertes Routing verwendet, so kann SANDMAN bereits vorhandene Cluster nutzen.

Jeder Knoten registriert die Dienste, die er anbietet, beim Clusterhead, indem er diesem entsprechende Beschreibungen zusendet. Da der Clusterhead nun in der Lage ist, Diensterkennungsanfragen für alle anderen Knoten des Clusters zu beantworten, können sich diese schlafenlegen. Zuvor spricht jeder Knoten die Schlafdauer mit dem Clusterhead ab, wie in Abbildung 2 dargestellt.

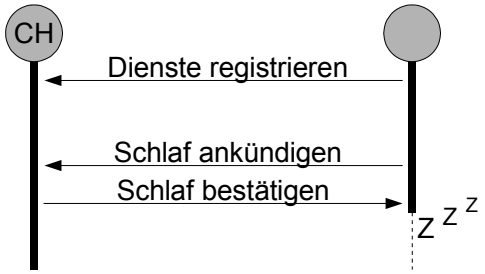

Abbildung 2: Registrieren und Schlafen bei SANDMAN

Möchte ein Knoten – der *Klient* – einen Dienst nutzen, so sendet er zunächst eine Diensterkennungsanfrage an den Clusterhead und bekommt von diesem jene Knoten genannt, die den gesuchten Dienst bieten.

Damit der Klient einen Dienst wirklich nutzen kann, muss der zugehörige Knoten, fortan *Dienstknoten* genannt, natürlich wach sein. Den nächsten Aufwachzeitpunkt bekommt der Klient ebenfalls vom Clusterhead mitgeteilt.

Würden mehrere Klienten nach dem Aufwachen eines Dienstknotens ihre Anfragen sofort senden, so würden diese kollidieren. Daher wartet jeder Klient nach dem Aufwachen des Dienstknotens eine zufällig gewählte Zeit T_{Bo}, bevor er seine Anfrage sendet. Während ein Klient seine Anfrage sendet, halten die anderen Klienten den Zeitgeber zu T_{Bo} an. Dadurch läuft bei einem Klienten die Zeit T_{Bo} nur ab, wenn das Übertragungsmedium momentan frei ist. Dieses Verfahren wird *Backoff-Mechanismus* genannt und kommt auch in vielen Media Access Control-Algorithmen zum Einsatz.

Der Dienstknoten beantwortet die Anfragen daraufhin – kann währenddessen aber weitere entgegennehmen. Nach dem Senden der letzten Antwort wartet der Dienstknoten noch eine

gewisse Zeit T_w, bevor er wieder mit dem Clusterhead über die nächste Schlafdauer verhandelt und sich anschließend schlafenlegt. Dasselbe gilt auch, wenn ein Dienstknoten nach dem Aufwachen in der Zeit T_w überhaupt keine Anfrage erhalten hat. In Abbildung 3 wird eine Diensterkennung und anschließende -nutzung dargestellt.

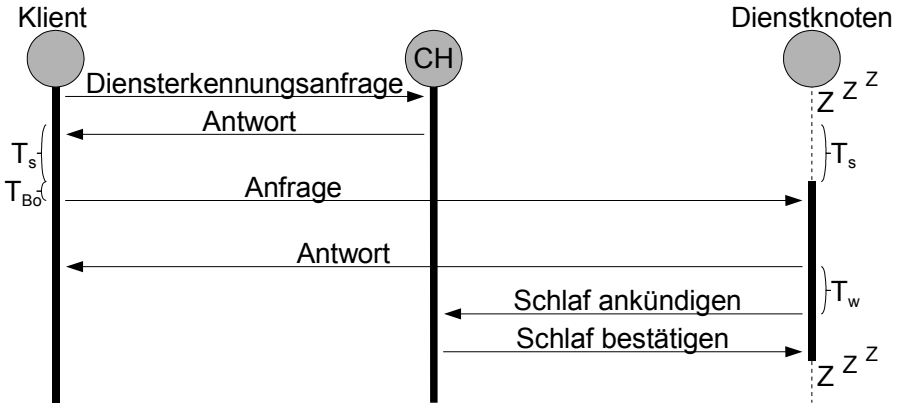

Abbildung 3: Diensterkennung und -nutzung bei SANDMAN

Die Zeit T_w dient nicht nur dazu geringe Uhrenabweichungen und den Backoff beim Senden von Anfragen zu kompensieren, sondern ist auch für die rasche Bearbeitung von Folgeanfragen wichtig. Eine Folgeanfrage ist eine Anfrage, die ein Klient direkt auf eine Antwort hin an denselben Dienst sendet. Wird T_w recht kurz gewählt, so besteht die Gefahr, dass eine Folgeanfrage den Dienst zu spät erreicht. Der Klient muss in diesem Fall bis zur nächsten Wachphase des Dienstes warten. Um unangenehme Wartezeiten zu vermeiden, sollte T_w so gewählt werden, dass Folgeanfragen den Dienst innerhalb derselben Wachphase erreichen.

Wird ein Dienst während seiner Wachphase überhaupt nicht genutzt, so kann die Zeit T_w allerdings unnötig lang sein. Eine Möglichkeit ist, zu Beginn einer Wachphase eine kürzere Zeit T'_w, nur abhängig vom Backoff-Mechanismus, zu warten.

Eine andere Möglichkeit wird in SEDUCE-2, dem Vorgänger von SANDMAN verwendet [Angstm 2003]. Dort registrieren alle Klienten jede Dienstnutzung beim Clusterhead. Wacht ein Dienstknoten auf, so fragt er zunächst den Clusterhead, ob Klienten ihn nutzen möchten. Falls nicht, kann der Dienstknoten sofort wieder schlafen. Da alle Dienstnutzungen beim Clusterhead angemeldet werden, kann dieser jedem Klienten ein Zeitfenster für das Senden der Anfrage zuteilen. Dadurch werden Kollisionen verhindert.

Viele energieeffiziente Algorithmen für Ad-hoc-Netze verwenden gemeinsame Wachphasen aller Knoten eines Clusters [IEEE 802.11, WeHeEs 2002, DamLan 2003]. Das vereinfacht zwar die Synchronisation, da die Knoten immer gleichzeitig aufwachen, erhöht aber auch die Wahrscheinlichkeit von Nachrichtenkollisionen. Lange Schlafzeiten senken außerdem direkt die mittlere Übertragungsrate.

Da es bei SANDMAN keine clusterweiten Schlaf- und Wachphasen gibt, senkt SANDMAN weder die mittlere Übertragungsrate, noch erhöht es die Wahrscheinlichkeit von Kollisionen in diesem Maße.

3.3 Allgemeine Anforderungen

Die Anforderungen an ein Verfahren zur energieeffizienten Dienstnutzung scheinen zunächst recht einfach: Es soll möglichst wenig Energie benötigen, sich aber sonst nicht von „normaler“ Dienstnutzung unterscheiden. Diese zentrale Anforderung beinhaltet eine Reihe weiterer:

- *Lange Schlafphasen:* Je länger die Knoten schlafen, desto weniger Energie benötigen sie. Dabei muss natürlich auch das Verhältnis von Wach- zu Schlafdauer berücksichtigt werden. Lange Schlafphasen reduzieren aber auch die Zahl der Wechsel zwischen Wach- und Schlafmodus, die mit zusätzlichen Energiekosten, den Transitionskosten, verbunden sein können.
- *Kurze Wartezeiten mit angeschalteter Netzwerkschnittstelle:* Das gilt für Knoten, die momentan als Dienstgeber arbeiten, wie für Klienten. Sobald ein Knoten aufwacht oder eine Anfrage gesendet hat, soll er möglichst rasch mit den benötigten Informationen versorgt werden. Letzteres kann eine Anfrage beziehungsweise eine Antwort sein, aber auch eine Benachrichtigung, dass er in den Schlafmodus wechseln kann.
- *Rasche Erreichbarkeit:* Das Abschalten der drahtlosen Netzwerkschnittstelle hat direkten Einfluss auf die Erreichbarkeit eines Knotens im Netz. Nachrichten an ihn müssen gegebenenfalls verzögert werden. Für eine schnelle Diensterbringung sollten Anfragen und Antworten rasch zugesendet und entsprechend Verzögerungen wegen Schlafphasen minimiert werden.
- *Gleichmäßiger Energieverbrauch:* Die Energievorräte der Knoten soll möglichst gleichmäßig aufgebraucht werden. Kein Knoten des Netzes darf auf Dauer übermäßig belastet werden. Der vorzeitige Ausfall einzelner Knoten soll vermieden werden, da je nach Position der Knoten solche Ausfälle eine Partitionierung des Netzes verursachen können. Netzpartitionierungen verhindern die Zusammenarbeit von Knoten, die in verschiedenen Partitionen liegen.
- *Wenig Nachrichten:* Die Anzahl von Nachrichten soll so gering wie möglich sein. Das gilt für Nachrichten zur Dienstnutzung wie für Kontrollinformationen.
- *Kurze Nachrichten:* Alle Nachrichten sollen möglichst kurz gefasst sein. Insbesondere die Kontrollinformationen sollen so knapp wie möglich gehalten werden.
- *Skalierbarkeit des Netzes:* Das Netz soll ohne Leistungseinbußen erweiterbar sein und sich an den aktuellen Bedarf an Nachrichtendurchsatz und Übertragungsrate anpassen.

Leider stehen viele dieser Anforderungen in Konflikt. Lange Schlafphasen vergrößern zum Beispiel die Wartezeiten und umgekehrt. Ein geeignetes Verfahren muss solche Konflikte so weit wie möglich entschärfen.

3.4 Anforderungen aus der Aufgabenstellung

In dieser Arbeit soll ein Verfahren zur energieeffizienten Dienstnutzung als Erweiterung beziehungsweise Variante von SANDMAN [ScBeRo 2004] untersucht werden. SANDMAN wurde

zur energieeffizienten Diensterkennung entwickelt. Allerdings lassen sich bezüglich des Energieverbrauchs die Erkennung und die Nutzung von Diensten nicht klar trennen, da Wartezeiten eines Dienstknotens nicht genau einem Aspekt, Diensterkennung *oder* Dienstnutzung, zugerechnet werden können.

Die hier betrachteten Verfahren zur energieeffizienten Dienstnutzung sollen wie SANDMAN auf der Bildung von Clustern mit je einem Clusterhead basieren.

Die letzte Anforderung aus der Aufgabenstellung ist die Verwendung des Clusterheads als Stellvertreter für die Dienstknoten. Ein Klient sendet seine Anfrage immer an den Clusterhead und erhält von diesem auch die Antwort. Dieser braucht die Antwort aber nicht selber zu ermitteln, sondern kann die Anfrage an einen geeigneten Dienstknoten weiterleiten. Er versteckt den eigentlichen Dienstgeber vor dem Klienten, wie in Abbildung 4 gezeigt.

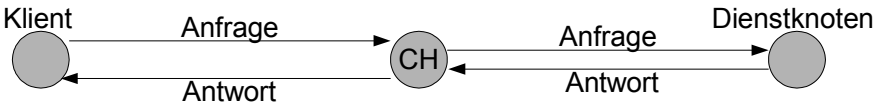

Abbildung 4: Clusterhead als Dienststellvertreter

4 Theoretische Analyse

In diesem Kapitel werden Vor- und Nachteile energieeffizienter Dienstnutzung mit dem Clusterhead als Dienststellvertreter untersucht und bewertet. In diesem Zusammenhang werden viele Variationsmöglichkeiten für ein entsprechendes Verfahren aufgezeigt.

Zuvor wird ein Überblick gegeben wo und wie in Kommunikationssystemen prinzipiell Energie gespart werden kann.

4.1 Verwendung von Anwendungswissen zur Energieeffizienz

Ad-hoc-Netze können nach dem OSI[3]-Referenzmodell in sieben Schichten eingeteilt werden. Diese sind in Tabelle 1 dargestellt.

7. Anwendung	(Application)
6. Darstellung	(Presentation)
5. Sitzung	(Session)
4. Transport	(Transport)
3. Vermittlung	(Network)
2. Sicherung	(Data Link)
1. Bitübertragung	(Physical)

Tabelle 1: Schichten des OSI-Referenzmodells

[3]Open Systems Interconnection

Energie kann auf verschiedenen Schichten gespart werden, wie im Folgenden einzeln erläutert.

Die Bitübertragungsschicht lässt sich durch die Wahl geeigneter elektronischer Komponenten energieeffizient gestalten [Atheros 2003].

Die Sicherungsschicht hat vor allem die Aufgabe der Media Access Control, kurz MAC. Viele Arbeiten [SinRag 1998, IEEE 802.11, EunVai 2002, DamLan 2003] befassen sich mit energieeffizienten MAC-Algorithmen für drahtlose Netze.

Die Vermittlungsschicht leitet einzelne Datenpakete durch das Netz. Im Englischen wird sie statt Network-Layer daher auch Routing-Layer genannt. Für diese Schicht existieren ebenfalls eine Reihe energieeffizienter Algorithmen [BeJa+ 2001, YaHeEs 2001, SriChi 2002, IkiOga 2003, YaBi+ 2003].

Die Schichten vier bis sechs sind für die Energieeffizienz weniger bedeutsam. Durch kurze kompakte Nachrichten und das Reduzieren von Kontrollinformationen lässt sich das Datenaufkommen minimieren und damit Energie sparen. Die meisten Protokolle für diese Schichten berücksichtigen diese Anforderungen bereits. Trotzdem müssen die Schichten vier bis sechs beim Entwurf energieeffizienter Algorithmen berücksichtigt werden. So verwendet zum Beispiel TCP[4] – ein gängiges Protokoll der Transportschicht – Timeouts, um Störungen zu entdecken. Wird nun die Netzwerkschnittstelle durch den energieeffizienten Algorithmus einer anderen Schicht für längere Zeit abgeschaltet, so könnte TCP voreilig meinen, die Verbindung sei abgebrochen.

SANDMAN und auch die hier betrachteten Varianten setzen auf der Anwendungsschicht an. Die Verwendung von Wissen um die Anwendungen verspricht größtmögliche Flexibilität. Die Fülle der möglichen Anwendungen aber macht gezielte Maßnahmen schwierig, da fast unüberschaubar viele Fälle und Szenarien unterschieden werden müssen. In diesem Spannungsfeld stehen alle weiteren Ideen und Bewertungen zur energieeffizienten Dienstnutzung mit dem Clusterhead als Stellvertreter der Dienste.

Energieeffizienz lässt sich vor allem durch Zusammenarbeit über Schichtengrenzen hinweg erreichen. Auch SANDMAN arbeitet nicht nur auf Anwendungsebene. Die Vermeidung von Anfragekollisionen durch den Backoff-Mechanismus ist eine Aufgabe der Sicherungsschicht. Auf der anderen Seite kann SANDMAN Cluster und Clusterhead von der Vermittlungsschicht verwenden, falls diese auf Clusterbildung basiert.

4.2 Vorteile des Clusterheads als Dienststellvertreter

Durch die Verwendung des Clusterheads als Stellvertreter der eigentlichen Dienste ergeben sich eine Reihe von Vorteilen, verglichen mit normaler Dienstnutzung und SANDMAN. Sie werden nun im Einzelnen genannt und erläutert.

Senden von Anfragen ist einfacher Das Senden von Anfragen ist aus Sicht des Klienten deutlich einfacher. Bei der ursprünglichen Variante von SANDMAN muss der Clusterhead dem Klienten mitteilen, wann der einzelne Dienstknoten aufwacht. Der Klient muss dann genau im passenden Moment seine Anfrage senden.

[4] Transmission Control Protocol

Wird die Anfrage nun über den Clusterhead als Stellvertreter gesendet, so entfällt diese komplizierte Prozedur aus Sicht des Klienten. Der Rhythmus von Schlaf- und Wachphasen ist vor dem Klienten versteckt.

Die Wahrscheinlichkeit für Kollisionen beim Senden von Anfragen ist bei SANDMAN sehr hoch, da die Anfragen innerhalb der kurzen Wachphase des Dienstknotens gesendet werden. Zwar senkt SANDMAN diese Wahrscheinlichkeit durch den Backoff-Mechanismus, aber mit dem Clusterhead als Dienststellvertreter ist sie noch kleiner, da die Klienten ihre Anfrage jederzeit senden können. Ein Backoff-Mechanismus wird nicht benötigt.

Vor allem bei Anfragen, die nicht beantwortet werden, macht sich die Möglichkeit bezahlt, sofort senden zu können. Bei SANDMAN muss das Senden einer solchen Anfrage solange verzögert werden, bis der Empfänger aufwacht. Ohne entsprechende Unterstützung wird damit auch die zugehörige Anwendung des Klienten blockiert. Mit dem Clusterhead als Dienststellvertreter kann der Klient seine Anfrage sofort senden und weiterarbeiten.

Klient kann schlafen Die Entkopplung von Dienst und Klient durch den Clusterhead ermöglicht auch dem Klienten, regelmäßig in eine Schlafphase zu wechseln. Der Clusterhead kann Antworten an den Klienten puffern, bis dieser wieder aufwacht.

Da der Clusterhead die Wartezeiten bei den verschiedenen Diensten kennt und durch Messungen sogar die Dauer der Diensterbringung abschätzen kann, kann der Clusterhead einen Klienten nach Empfang einer Anfrage sofort über die vermutliche Bearbeitungsdauer informieren und entsprechend schlafen schicken.

Beliebige Schlaf- und Wachzeiten Fungiert der Clusterhead als Stellvertreter für die Dienstknoten, so vereinfacht sich das Verfahren auch aus Sicht der einzelnen Dienstknoten. Sie können ihre Schlaf- und Wachzeiten nun beliebig wählen, da sie nur auf einen Knoten – den Clusterhead – Rücksicht nehmen müssen. Kein anderer Knoten braucht ihre Schlaf- und Wachzeiten zu kennen.

Ein Dienstknoten muss nun auch nicht zu Beginn einer Wachphase eine gewisse Zeit auf Anfragen beliebiger Klientknoten warten. Da einzige Klient eines Dienstknotens der Clusterhead ist, kann dieser den Dienstknoten sofort nach dessen Aufwachen über ausstehende Anfragen informieren oder ihm mitteilen, dass er sich sofort wieder schlafen legen kann.

Dynamische Wahl des Dienstknotens Bieten mehrere Knoten den gleichen Dienst, so kann der Clusterhead für jede Anfrage an diesen Dienst entscheiden, welcher Knoten sie beantworten soll.

Der Clusterhead kann den Zustand und die Auslastung aller Knoten in seinem Cluster einschätzen und nach verschiedenen Kriterien den optimalen Dienstknoten für jede Anfrage wählen. Denkbare Kriterien sind:

- *Latenz:* Vermutlich beantwortet der Dienstknoten, der als nächstes aufwacht, die Anfrage aus Sicht des Klienten am schnellsten.
- *Energievorrat:* Wählt man den Dienstknoten mit dem größten Energievorrat, so wird die Lebensdauer aller Knoten optimiert.

- *Auslastung:* Der Dienstknoten mit der geringsten Auslastung unter den momentan wachen Knoten beantwortet eine Anfrage in der Regel besonders schnell.
- *Übertragungsrate:* Bei langen Anfragen oder beim Erwarten einer langen Antwort kann auch die Übertragungsrate des Dienstknotens ein Kriterium sein. Eine hohe Übertragungsrate senkt die Energiekosten je Byte und verringert auch die Bearbeitungszeit aus Sicht des Klienten.

Einfache Diensterkennung Wird der Clusterhead als Stellvertreter für die eigentlichen Dienste eingesetzt, so ist Diensterkennung sehr einfach. Auf entsprechende Anfragen eines Klienten muss der Clusterhead nur mit „Ja, einen solchen Dienst gibt es“ oder „Nein, einen solchen Dienst gibt es nicht“ antworten. Der Klient benötigt Informationen wie den zugehörigen Dienstknoten und dessen nächste Wachphase nicht mehr. Die kürzeren Antworten schonen auch Übertragungskapazitäten des Netzes.

Diensterkennungsanfragen können sogar ganz weggelassen werden. Der Klient sendet sofort die eigentliche Anfrage an den Clusterhead. Dieser beantwortet die Anfrage entweder, indem er sie an einen passenden Dienst weiterleitet, oder, indem er dem Klienten mitteilt, dass es einen Dienst dafür nicht gibt.

Kompakte Anfrage- und Antwortpakete Werden alle Anfragen an einen bestimmten Dienstknoten vom Clusterhead gesammelt, so lassen sie sich zu einem großen Paket zusammenfassen. Der Dienstknoten erhält dann nicht viele kurze Nachrichten mit je einer Anfrage, sondern eine große Nachricht mit vielen Anfragen, die er nun einzeln oder auch wieder in einem großen Paket beantworten kann. Bei Verwendung des Clusterheads als Stellvertreter für die Dienste muss sich die Zahl der Nachrichten daher nicht einfach verdoppeln.

Bei bestimmten Diensten kann der Clusterhead doppelte Anfragen löschen und die Antwort dann entsprechend verteilen. Ein Beispiel hierfür sind Anfragen an Sensoren. Möchten zum Beispiel fünf Knoten im selben Zeitraum wissen, wie hoch die Raumtemperatur momentan ist, so kann der Clusterhead die fünf Anfragen sammeln, den entsprechenden Sensorknoten einmal anfragen und die Antwort dann an alle fünf Klienten verteilen.

Entsprechendes gilt auch, wenn ein Dienst viele Knoten gleichzeitig über ein Ereignis informieren möchte. Er braucht die entsprechende Nachricht nur einmal an der Clusterhead zu senden; dieser kann sie dann entsprechend verteilen.

Einfacher Zugriff auf zusammengesetzte Dienste Manche Dienste werden von mehreren Knoten in Kooperation erbracht. Werden alle Anfragen an den Clusterhead gesendet, so vereinfachen sich solche Prozeduren aus Sicht der Klienten erheblich. *Ein* Knoten ist für alles zuständig. Ein Beispiel aus der Gegenwart ist IMAP.[5] Möchte ein Klient eine Email senden, so muss er sich zunächst an den SMTP[6]-Server wenden, der sie dann über das Internet versendet. Das anschließende Speichern der Email in einem Ordner „Gesendet“ erfolgt dann durch Übertragen der Email an den IMAP-Server. Haben beide Server den Clusterhead als

[5] Internet Message Access Protocol
[6] Simple Mail Transfer Protocol

Stellvertreter, so muss der Klient nur mit einem Knoten kommunizieren. Denkbar ist sogar, dass der Klient die Email nur einmal sendet und der Clusterhead sie dann an SMTP- und IMAP-Server verteilt.

Einfache Netztopologie Werden alle Anfragen und Antworten über den Clusterhead geleitet, so vereinfacht sich die Netztopologie erheblich. Während in der ursprünglichen Variante von SANDMAN zwei beliebige Knoten miteinander kommunizieren können, kommuniziert hier jeder Knoten ausschließlich mit dem Clusterhead. Der Clusterhead arbeitet folglich wie ein Router und entsprechend kann bei geeigneter Konfiguration auf das eigentliche Routing ganz verzichtet werden – auch wenn manche Knoten, wie in Abbildung 5, zwei Hops auseinander liegen.

Eine solche geeignete Konfiguration wird allerdings, wenn überhaupt, nur zeitweise bestehen. Wechselt der Clusterhead, so wird das eigentliche Routing in der Regel wieder benötigt.

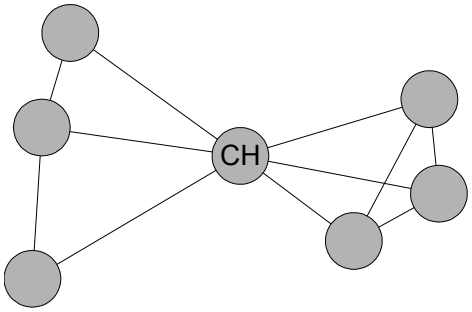

Abbildung 5: Clusterhead macht Routing überflüssig

Verbesserte Fehlertoleranz Der Clusterhead kann auf einfache Weise für Fehlertoleranz sorgen. Leitet er die Anfrage eines Klienten an einen Dienstknoten weiter und fällt dieser während der Diensterbringung aus, so kann der Clusterhead die Anfrage nochmals an einen anderen Knoten senden, der denselben Dienst bietet. Aus Sicht des Klienten hat die Beantwortung dieser Anfrage dann etwas länger gedauert, doch von der Fehlersituation und der Ausnahmebehandlung hat er nichts mitbekommen.

Der Clusterhead kann die Anfrage sogar parallel an mehrere Dienstknoten senden und aus deren Antworten eine sinnvolle Antwort bestimmen, die er an den Klienten weiterleitet.

Geringere Uhrenabweichungen Auch die Wahrscheinlichkeit von Fehlern bei der Synchronisation von Schlaf- und Wachphasen ist bei der Nutzung des Clusterheads als Dienststellvertreter geringer. Bei der ursprünglichen Variante von SANDMAN einigt sich jeder Dienstknoten mit dem Clusterhead über Schlaf- und Wachzeiten. Verzögerungen beim Senden der entsprechenden Nachrichten können für Abweichungen der Uhren sorgen. Auf Diensterkennungsanfragen hin werden die Schlaf- und Wachzeiten den Klienten mitgeteilt. Ein weiteres

Mal können durch Verzögerungen Abweichungen der Uhren entstehen. Man spricht von *Multi-Hop-Synchronisation*.

Dient der Clusterhead als Stellvertreter für die Dienste, so braucht er die Schlaf- und Wachzeiten der einzelnen Dienstknoten niemandem mitzuteilen. Ein solches Verfahren wird *Single-Hop-Synchronisation* genannt. Bei dieser Art der Synchronisation weichen die Uhren im Allgemeinen geringer ab.

4.3 Nachteile des Clusterheads als Dienststellvertreter

Natürlich bringt die Nutzung des Clusterheads als Stellvertreter für alle Dienste im Cluster auch eine Reihe von Nachteilen und Problemen mit sich, verglichen mit normaler Dienstnutzung und SANDMAN. Diese werden im Folgenden detailliert diskutiert und für die Probleme, falls möglich, Lösungsansätze gegeben.

Erhöhtes Nachrichtenaufkommen Ein wichtiger Nachteil ist die erhöhte Datenmenge, die übertragen wird. In erster Näherung muss man hier mit dem Faktor zwei rechnen, da jede Anfrage und jede Antwort doppelt übertragen wird. Wie bereits oben angedeutet, gibt es eine Reihe von Möglichkeiten, diesen Faktor zu senken.

- Der Clusterhead fasst Anfragen mehrerer Klienten, die an denselben Dienstknoten gesendet werden, zu einer Nachricht zusammen.
- Ebenso kann der Dienstknoten mit den Antworten verfahren.
- Gleiche Anfragen werden dem Dienst nur einmal gestellt.
- Clusterhead werden nur solche Knoten, die selber viele Dienste bieten und erbringen.

Die Energiekosten für das Senden und Empfangen spielen beim Clusterhead eine untergeordnete Rolle, da sie, wie in Kapitel 5.2 erklärt, nicht viel höher sind als das Mithören von Nachrichten.

Bei besonders großen Datenmengen aber stellt sich die Frage, ob der Weg über den Clusterhead sinnvoll ist.

Ein größeres Problem ist der Clusterhead als Rechner selbst. Er muss alle Anfragen zwischenspeichern und kann zum Engpass werden, wenn seine Puffer zu klein sind. Daher sollten nur Knoten mit ausreichender Speicherkapazität und mit Netzwerkschnittstellen mit hohen Übertragungsraten zum Clusterhead gewählt werden.

Allerdings schränken die eben genannten Anforderungen, und die Anforderung ein Clusterhead solle selber viele Dienste bieten und erbringen, die Menge der für die Aufgabe des Clusterheads in Frage kommenden Knoten ein. Dies macht eine gleichmäßige Belastung aller Knoten schwierig.

Größere Verzögerungen Durch die Übertragung der Anfragen und Antworten über den Clusterhead verzögern sich auch im günstigsten Fall – wenn der Dienstknoten gerade wach ist – alle Nachrichten. Dies wirkt sich in zweierlei Hinsicht aus:

1. Aus Sicht des Klienten dauert jede Diensterbringung etwas länger. Insbesondere wenn derselbe Dienst in kurzen Abständen mehrmals hintereinander genutzt wird und die eigentliche Diensterbringung sehr schnell verläuft, machen die zusätzlichen Verzögerungen einen großen Teil der benötigten Zeit aus.

2. Aus Sicht des Dienstknotens werden bei mehrfacher Nutzung des Dienstes durch denselben Klienten die Abstände zwischen Antwort und nächster Anfrage größer. Die ursprüngliche Variante von SANDMAN sieht ein Timeout vor, nachdem ein Dienst davon ausgehen kann, dass Klienten, die ihn mehrfach direkt hintereinander benötigen, alle Anfragen gestellt haben. Dieser Timeout, der je nach Dienst einen signifikanten Teil des Energieverbrauchs ausmacht, muss bei der hier beschriebenen Variante verlängert werden.

Da die Latenzen bei steigenden Übertragungsraten in etwa gleich bleiben, wird sich diese Problematik bei künftigen Entwicklungen verschärfen!

Unregelmäßige Bearbeitungszeiten Aus Sicht der Klienten beginnt jede Diensterbringung mit dem Senden der Anfrage an den Clusterhead und endet mit dem Empfangen der Antwort vom Clusterhead. Was dazwischen passiert, ist für den Klienten unsichtbar. Eine Anfrage kann je nach Schlafphase das eine Mal binnen kürzester Zeit und ein anderes Mal erst nach einer längeren Wartezeit beantwortet werden. Der Klient hat keine Möglichkeit, sich auf die Schlaf- und Wachphasen der Dienstknoten einzustellen, da er sie nicht kennt.

Damit können mehrere Anfragen nicht entsprechend der Aufwachzeitpunkte der Dienste sortiert und gestellt werden.

Kann ein Klient eine Anfrage an mehrere Clusterheads senden, so ist es ihm nicht möglich den Clusterhead zu bestimmen, über den seine Anfrage am schnellsten beantwortet wird. Es gibt zwei Möglichkeiten diese Probleme zumindest teilweise zu umgehen:

1. Ein Klient kann verschiedene Anfragen parallel stellen, das heißt, er sendet mehrere Anfragen direkt hintereinander, auch wenn die erste Anfrage noch nicht beantwortet wurde. Anschließend wartet er, bis er alle Antworten, in beliebiger Reihenfolge, erhalten hat. Diese Möglichkeit ist sogar einfacher als das Sortieren von Anfragen, wie es in der ursprünglichen Variante von SANDMAN möglich ist, bindet oft aber mehr Ressourcen.

   Stehen einem Klienten für eine Anfrage mehrere Clusterheads zur Verfügung, so macht das parallele Stellen der Anfrage und anschließende Warten auf die erste Antwort, aus Sicht des Energieverbrauchs, keinen Sinn, da die Anfrage entsprechend mehrfach beantwortet würde.

2. In der Antwort zu einer Diensterkennungsanfrage kann der Clusterhead die Klienten über die Mindestwartezeit, die sie zum jetzigen Zeitpunkt für einen bestimmten Dienst aufbringen müssen, informieren. Damit haben Klienten einen Anhaltspunkt, wann entsprechende Anfragen eigentlich bearbeitet würden.

Dienstknoten kennen die Klienten nicht Werden die Anfragen der Klienten ohne Zusatzinformationen vom Clusterhead an die Dienstknoten weitergeleitet, so muss der Dienstknoten zunächst davon ausgehen, dass die Anfrage wirklich vom Clusterhead stammt. Der Dienst sieht den eigentlichen Klienten nicht, weil der Clusterhead ihn vor ihm versteckt.

Für einige Dienste, zum Beispiel beim Abfragen von Sensorwerten, ist dies kein Problem – für sicherheitskritische Dienste aber ein großes. Die Absenderkennung kann in einem solchen Fall auch bei einem Netz, das fälschungssichere Kennungen bietet, nicht verwendet werden.

Ähnliches gilt auch für Ereignisse. Sie können spontan von einem Dienst an einen Klienten gesendet werden. Auch hier muss der Dienst explizite Informationen über seine Klienten halten, damit er diese benachrichtigen kann. Es gibt zwei Ansätze, diese Probleme zu umgehen:

1. Der Clusterhead teilt den Diensten in jeder Anfrage den eigentlichen Klienten mit. In vielen Fällen wird dies aber unnötig und damit Verschwendung von Ressourcen sein.

   Als Gegenmaßnahme können jene Dienste, die die eigentlichen Klienten kennen müssen, dies dem Clusterhead in der Dienstbeschreibung mitteilen, so dass dieser sich entsprechend darauf einstellen kann.

2. Die Dienste verwenden die Absenderkennung von Anfragen nicht, sondern verlangen sie gegebenenfalls explizit in ihren Nachrichtenformaten. Diese Vorgehensweise schafft allerdings neue Fehlerquellen in der Anwendungsentwicklung.

Keine Erkennung von Folgeanfragen In vielen Fällen hat eine Dienstnutzung die einfache Form „Anfrage – Diensterbringung – Antwort“ und ist damit vollständig beendet.

Es gibt allerdings auch eine Reihe von Diensten, bei denen die Diensterbringung von vorherigen Anfragen desselben Klienten abhängt. Ein einfaches Beispiel ist der Lesezugriff auf Dateien. Zunächst wird die Datei geöffnet. Anschließend kann mit entsprechenden Anfragen der Inhalt gelesen werden. Zum Abschluss muss die Datei explizit geschlossen werden.

Wird jede Operation – Öffnen, Lesen und Schließen – als separate Dienstnutzung durchgeführt, so muss der Dienst nicht nur den eigentlichen Klienten kennen, um ihm im Falle mehrerer gleichzeitig geöffneter Dateien die passenden Daten liefern zu können, sondern der Clusterhead muss auch garantieren, dass die Anfragen eines Klienten immer beim selben Dateidienst landen. Der Clusterhead muss eine Art *Sitzung*, nach dem Englischen oft *Session* genannt, verwenden.

Aufwändige Fehlerbehandlung Die Verwendung des Clusterheads als Dienststellvertreter eröffnet neue Fehlerquellen. Sind in der ursprünglichen Variante von SANDMAN bei der eigentlichen Diensterbringung nur zwei Knoten – Klient und Dienst – beteiligt, so sind es bei den hier diskutierten Varianten und Erweiterungen drei. Die Wahrscheinlichkeit, dass eine Dienstnutzung wegen eines Knotenfehlers scheitert ist größer als bei SANDMAN.

Allerdings hat der Clusterhead auch in der ursprünglichen Variante von SANDMAN eine tragende Rolle, so dass ein Ausfall dieses Knotens entsprechend abgefangen werden muss. Die Ausnahmebehandlungen müssen nun von der Diensterkennung auf die Dienstnutzung übertragen werden.

Ein Klient muss unterscheiden können, ob der eigentliche Dienst oder der Clusterhead ausgefallen ist, wenn er statt einer Antwort eine Fehlermeldung erhält.

Es stellt sich die Frage, wie bei Ausfall des Clusterheads mit Anfragen umgegangen wird, die momentan bearbeitet werden. Es gibt drei Möglichkeiten:

1. Sobald ein Dienstknoten erkennt, dass der Clusterhead ausgefallen ist, bricht er die Bearbeitung aller Anfragen ab und wartet, bis ein neuer Clusterhead wieder Anfragen liefert.

2. Der Dienstknoten bearbeitet die Anfragen, hält die Antworten aber zurück, bis im Zuge einer Ausnahmebehandlung ein Klient seine Anfrage erneut über einen neuen Clusterhead sendet. Dann kann der Dienst diese sofort beantworten.

   Bieten mehrere Knoten den gleichen Dienst an, so ist aber nicht garantiert, dass erneut gesendete Anfragen wieder an denselben Dienstknoten weitergeleitet werden.

3. Der Dienst bearbeitet die Anfragen und sendet die Antworten über einen neuen Clusterhead oder direkt zurück. Die Klienten verwenden eine entsprechende Ausnahmebehandlung, so dass sie nicht irritiert sind, wenn die Antwort von einem anderen Knoten kommt, als der, an den sie die Anfrage gesendet hatten.

Die Entkopplung von Dienst und Klient über den Clusterhead hat auch neue Folgen für den Fall, dass ein Klient- oder Dienstknoten ausfällt. Der Clusterhead muss Sessions, an denen dieser Knoten beteiligt war, beenden und den entsprechenden Partnerknoten darüber informieren. Das gilt auch schon für den Fall, dass ein Dienstknoten explizit Informationen über einen Klienten speichert, der den Cluster durch einen Fehlerfall abrupt verlassen hat.

Clusterhead muss Anfragen einsehen Viele der oben genannten Konzepte und Optimierungen basieren darauf, dass der Clusterhead die Anfrage direkt einsehen kann. So kann er doppelte Anfragen verhindern, Dienste optimal auslasten oder Sessions verwalten. Allerdings ist dies gerade bei spontan vernetzten und potentiell unbekannten Rechnern ein mögliches Sicherheitsrisiko. Viele Anfragen werden nicht für die Augen Dritter bestimmt sein.

Verschlüsselte Anfragen müssen daher zusätzlich unverschlüsselt jene Informationen enthalten, die der Clusterhead zur Bestimmung des passenden Dienstknotens benötigt. Die Nachrichtenformate werden komplexer.

Aufwändiger Wechsel des Clusterheads Auch der Wechsel des Clusterheads ist bei einer Verwendung als Dienststellvertreter aufwändiger als bei der ursprünglichen Variante von SANDMAN.

Aus Sicht eines Klienten migriert der Dienst, mit dem er eben zusammengearbeitet hat und aus Sicht eines Dienstes migriert der Klient.

Der neue Clusterhead sollte nicht nur das Verzeichnis der Dienstbeschreibungen übernehmen, sondern auch vorhandene Sessions zwischen Klienten und Dienstknoten übernehmen.

Clusterhead benötigt mehr Rechenleistung Der Clusterhead benötigt nicht nur, wie oben beschrieben, genügend Speicherkapazität für die Pufferung von Nachrichten, sondern auch eine ausreichende Rechenleistung. Im Vergleich zur ursprünglichen Variante von SANDMAN hat er deutlich mehr Informationen zu verwalten. Konzepte wie Sessions, Duplikaterkennung oder das Führen von Metriken zur Lastverteilung machen nicht nur die Dienstbeschreibungen aufwändiger, sondern bringen für den Clusterhead viele neue Verwaltungsaufgaben mit sich. Damit der Clusterhead nicht zum Engpass wird und damit die Skalierbarkeit des Clusters behindert, muss er über eine entsprechende Rechenleistung verfügen.

Auf Grund dieser Anforderungen kommen in vielen Fällen nicht alle Knoten für die Aufgabe des Clusterheads in Frage. Damit aber schwindet die Möglichkeit, die Knoten bezüglich des Energieverbrauchs gleichmäßig zu belasten.

Unnötige Multi-Hop-Verbindungen Die Nutzung von Diensten muss sich nicht auf den eigenen Cluster beschränken. Erfolgt die Nutzung eines Dienstes in einem entfernten Cluster über dessen Clusterhead, so kann dieser aus Sicht der Verbindung zwischen Klient und Dienstknoten eine besonders ungünstige Position haben.

Abbildung 6 zeigt ein Beispiel, in dem statt einer Single-Hop-Verbindung drei Hops verwendet werden.

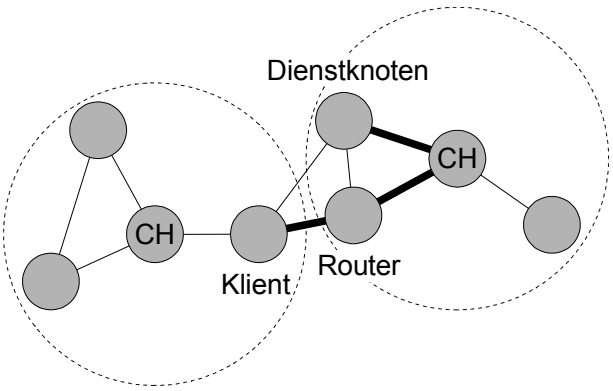

Abbildung 6: Unnötige Multi-Hop-Verbindung – drei Hops statt einem

4.4 Bewertung der Vor- und Nachteile

Eine Abwägung der genannten Vor- und Nachteile bei einer Nutzung des Clusterheads als Dienststellvertreter ist schwierig.

Das Prinzip besticht zunächst durch seine Einfachheit für die Klienten. Im Licht der beiden Varianten, die bei der Entwicklung von SANDMAN in [Angstm 2003] diskutiert wurden, scheint der Schritt zum Clusterhead als Dienststellvertreter nur logisch. Auch die Möglichkeit, den Klienten während der Diensterbringung schlafen zu legen, ist von SANDMAN aus der nächste logische Schritt.

Leider gelten aber viele der genannten Vorteile nur für bestimmte Dienste oder in speziellen Szenarien. Es lässt sich nicht abschätzen, in wie vielen Fällen Möglichkeiten wie die dynamische Wahl des Dienstknoten, optimierte Anfragen oder einfacher Zugriff auf zusammengesetzte Dienste wirklich verwendet werden können.

Für manche Nachteile gilt aber dasselbe. Unnötige Multi-Hop-Verbindungen, aufwändigerer Wechsel des Clusterheads und das Nichterkennen von Folgeanfragen sind Nachteile, die nur bei einigen Diensten oder in wenigen Szenarien auftreten.

Viele der genannten Probleme können durch entsprechende Mechanismen behoben werden. Daher muss eine Nutzung des Clusterheads als Dienststellvertreter in realen Anwendungen erheblich durch entsprechende Software unterstützt werden. Auch der große Vorteil, dass der Dienstknoten beliebig gewählt werden kann, benötigt erhebliche Unterstützung durch Software, da der Clusterhead dazu umfangreiche Metriken führen sollte, will er den optimalen Dienstknoten für eine Anfrage bestimmen.

Soll ein solches Verfahren sinnvoll eingesetzt werden, so müssen zwei Bedingungen erfüllt sein:

1. Die Mehrzahl der Dienste muss ohne oder mit nur geringem Zusatzaufwand mit dem Clusterhead als Dienststellvertreter verwendet werden können.

2. Ein durchschnittlicher Knoten muss die Aufgabe des Clusterheads übernehmen können, ohne wegen zu geringer Rechenleistung oder Speicherkapazität zum Engpass zu werden.

Ein geeignetes Szenario für die Verwendung des Clusterheads als Dienststellvertreter ist ein System zur Pflege von Pflanzen [Zitter 2004, Kap. 2]. Es besteht aus Sensoren für Feuchtigkeit, Helligkeit und Temperatur und Aktoren wie Berieselungsanlagen und elektrischen Rollladen- und Fensterhebern. Ein solches System kann fest konfiguriert oder, wie hier angenommen, spontan vernetzt werden. Das System bildet ein Ad-hoc-Netz, in dem die Knoten ausnahmsweise nicht oder kaum mobil sind.

Das Pflanzenpflegesystem erfüllt die erste der eben formulierten Bedingungen, da weder Aktoren noch Sensoren für die Erbringung ihrer Dienste besondere Anpassungen an Klienten oder die Umgebung benötigen. Aktoren und Sensoren lassen sich über sehr einfache Schnittstellen wie `gieße(int ml)`, `gib_Temperatur()` oder `öffne_Fenster()` ansprechen.

Die zweite Bedingung ist ebenfalls erfüllt, da die Anzahl und Länge der Nachrichten gering ist und der Clusterhead kaum Verwaltungsinformationen in seiner Funktion als Dienststellvertreter zu speichern braucht.

5 Verwandte Arbeiten

Energieverbrauch und energieeffiziente Algorithmen in Ad-hoc-Netzen sind Thema einer ganzen Reihe von Arbeiten. Im Folgenden wird ein Einblick in die hier wichtigsten gegeben.

5.1 Energieverbrauch von Computern in drahtlosen Netzen

Messergebnisse in [StGa+ 1996] zeigen, dass die Funknetzwerkkarte bei Minicomputern der größte Energieverbraucher ist. [FeeNil 2001, JoSi+ 2001, SiBe+ 2000, KraBal 2002] bestäti-

gen dieses Ergebnis. Werden keine Energiesparmaßnahmen ergriffen, so macht der Funknetzwerkzugang zwischen 10% und 50% des gesamten Energieverbrauchs aus.

Atheros Communications hat in [Atheros 2003] eigene Funknetzwerkkarten mit denen anderer Hersteller in gängigen Laptops verglichen. Dabei wurde nicht nur die Leistungsaufnahme der entsprechenden Funknetzwerkkarte, sondern auch die des gesamten Laptops gemessen. Die Messung berücksichtigt also auch die zusätzliche Leistungsaufnahme von Hauptplatine und Prozessor zur Unterstützung des Funkbetriebs. Ein typischer Wert für diesen Anteil an der Gesamtleistung wird in Abbildung 7 genannt.

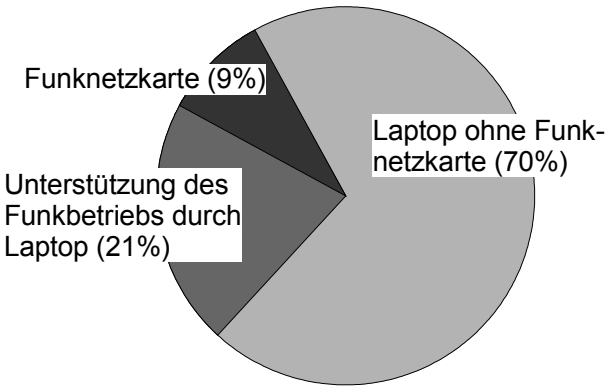

Abbildung 7: Leistungsaufnahme der verschiedenen Komponenten eines Laptops

[Atheros 2003] gibt sechs Hinweise zur Energieeffizienz von Funknetzhardware und Treibersoftware:

- Je höher die Übertragungsrate desto geringer der Energieverbrauch je übertragener Datenmenge.
- Die elektronischen Komponenten der Funknetzkarte haben einen erheblichen Einfluss auf den Energieverbrauch.
- Moderne Protokolle zwischen Netzwerkkarte und Hauptplatine, wie Direct Memory Access (DMA), sind sehr energieeffizient.
- So genannte *Mobile-Prozessoren*[7] haben eine geringere Leistungsaufnahme als gewöhnliche Prozessoren und benötigen entsprechend weniger Energie bei allen Aktivitäten des Hauptprozessors, die zur Nutzung eines Funknetzes benötigt werden.
- Schlanke Treibersoftware reduziert den Energieverbrauch des Hauptprozessors für Funknetzaktivitäten.
- Spezielle Hardware zum Ver- und Entschlüsseln der Funknetzdaten verbraucht deutlich weniger Energie als eine entsprechende Softwareimplementierung

[7]Fast alle Laptop-Hersteller bieten heute Modelle mit Mobile-Prozessoren, die Intel Centrino mobile- oder Mobile AMD-Technologie verwenden.

Tabelle 2 zeigt typische Werte für die zusätzliche Leistungsaufnahme P, die durch Betrieb einer Mini PCI WLAN Card an einem Laptop mit Pentium-M-Prozessor bei Verwendung des IEEE Standards 802.11 entsteht. PSM bezeichnet in Tabelle 2 den Power Save Mode von IEEE 802.11.

Aktion	P in $Watt$
Suche nach Funknetz	0,44 – 1,24
Leerlauf ohne PSM	1,14 – 1,82
Leerlauf mit PSM	0,19 – 0,91
Empfang von Daten	2,60 – 3,50
Senden von Daten	2,45 – 3,35

Tabelle 2: Leistungsaufnahme einer typischen Funknetzkarte

Der Durchsatz liegt dabei zwischen 4 und 22 Megabit pro Sekunde. Je übertragener Datenmenge unterscheiden sich die Kombinationen aus Hard- und Software im Energieverbrauch um bis zu Faktor fünfzehn.

5.2 Analysen und Modelle zu Sende- und Empfangskosten

Im Folgenden werden Messungen und mathematische Modelle zum Energieverbrauch von Funknetzkarten betrachtet. Zu den Messungen gibt es eine Fülle von Arbeiten und Herstellerangaben. Im Bereich der mathematischen Modelle lassen sich zwei Klassen unterscheiden:

- *Einfache Modelle:* Sie werden direkt aus Messungen gewonnen und sind auf einen einzelnen erfolgreichen Sende- oder Empfangsvorgang ausgelegt.
- *Komplexe Modelle:* Sie sind für aufwändige Szenarien geeignet und ermöglichen zusätzlich Aussagen über Kollisionen, Wartezeiten und Verzögerungen.

Energieverbrauch auf Basis von Paketen In [FeeNil 2001] wird ein einfaches Modell für das Senden, Empfangen oder Mithören eines Pakets nach dem IEEE Standard 802.11 entwickelt.

Ungewolltes Mithören des Funkverkehrs einer Punkt-zu-Punkt-Verbindung oder einer Multicast-Nachricht zwischen anderen Knoten ist gängiges Geschehen in Funknetzen. Es kostet den Knoten, der mithört, in der Regel zusätzlich Energie. Nur Funknetzkarten mit einer Übertragungsrate bis 2 Mbit/s können bei einer nicht an sie bestimmten Nachricht nach dem Empfang des Nachrichtenkopfes kurz in einen Schlafmodus wechseln und damit sogar Energie sparen.

Je nach Position des Mithörers kann dieser auch nur die Empfangsbestätigung oder die Nachricht ohne Empfangsbestätigung empfangen.

Die lineare Formel des Modells von [FeeNil 2001] lautet:

$$W = m_A \cdot l + b_A$$

Die Parameter und Variablen haben folgende Bedeutung:

- W bezeichnet die Energiekosten in Joule.
- m_A in Joule/Byte und b_A in Joule sind Konstanten, die je Netzwerkkarte und Aktion aus Messungen bestimmt werden. Als Aktionen werden Senden, Empfangen oder Mithören eines Unicasts mit Empfangsbestätigung beziehungsweise eines Broadcasts bezeichnet. Beim Mithören einer Nachricht wird noch unterschieden, ob die Nachricht ausgewertet oder sofort gelöscht wird.
- l bezeichnet die Länge des Pakets in Byte. Dabei werden nur die Nutzdaten berücksichtigt. Die Länge des Nachrichtenkopfes ist in b_A mit einberechnet.

Stochastische Analyse von IEEE 802.11 In [ZanPel 2004] wird eine präzise mathematische Analyse des Protokolls nach IEEE 802.11 für ein Ad-hoc-Netz beschrieben. Unterschieden wird dabei zwischen dem *Basic Access Mode* und dem Zugriff über *RTS/CTS* auf das Funknetz.

Beim Basic Access Mode [IEEE 802.11] werden die eigentlichen Nachrichten ohne Vorankündigung gesendet. Jeder potentielle Sender startet nach einer Funkstille der Dauer T_{DIFS}[8] einen Backoff-Mechanismus. Hat nach der zufällig gewählten Zeit noch keine andere Übertragung begonnen, so sendet der Knoten seine Nachricht. Nach der Übertragung bestätigt der Empfänger die Nachricht durch Senden eines Acknowledgments, kurz ACK.

Der Zugriff auf das Funknetz mit RTS/CTS [IEEE 802.11] kann nur für Unicast-Nachrichten erfolgen. Vor dem Senden der eigentlichen Nachricht reserviert der potentielle Sender nach einer Funkstille der Dauer T_{DIFS} und einer zufällig gewählten Zeit das Übertragungsmedium durch Senden von „Request to Send“, kurz RTS. Der Empfänger antwortet darauf mit „Clear to Send“, mit CTS abgekürzt. Dadurch wissen auch Knoten, die nur in der Reichweite des Empfängers liegen vorab von der anstehen Übertragung und wie lange sie Funkstille wahren müssen, um nicht zu stören. Anschließend überträgt der Sender die eigentliche Nachricht und der Empfänger bestätigt sie mit ACK.

Trotz dieser Mechanismen können in IEEE 802.11 Kollisionen auftreten. Mit deren Wahrscheinlichkeit werden in [ZanPel 2004] Erwartungswerte für die verschiedenen Wartezeiten im Protokoll ermittelt. Daraus und mit Hilfe dreier funkkartenspezifischer Leistungsparameter, zum Beispiel aus den Messergebnissen in [FeeNil 2001] berechenbar, wird der durchschnittliche Energieverbrauch für das erfolgreiche Senden eines Pakets in Abhängigkeit von der Anzahl der Knoten und einiger anderer Parameter bestimmt.

5.3 Media Access Control

Bereits in [StGa+ 1996], wo zwei PDAs von Apple und Sony und diverse Funknetzwerkkarten miteinander verglichen wurden, erkannte man, dass der Energieverbrauch beim Mithören von Nachrichten nicht wesentlich kleiner ist als beim Empfangen von Nachrichten. Bei typischer Verwendung eines entsprechenden Gerätes wird daher durch das Mithören in der Summe wesentlich mehr Energie verbraucht als durch das Senden und Empfangen. Daher empfehlen die Autoren von [StGa+ 1996] regelmäßiges Abschalten der Funknetzschnittstelle, sobald diese

[8] Distributed (coordination function) interframe space

nicht mehr gebraucht wird. Auf dieser Erkenntnis basieren nicht nur die folgenden Algorithmen zum Media Access Control, sondern auch die folgenden Algorithmen zum Routing und jene, die auf Anwendungsebene arbeiten. Zu den Letztgenannten gehört auch SANDMAN.

Heute existiert eine große Auswahl an entsprechenden Protokollen. Die meisten dieser Arbeiten setzen dabei auf der Media-Access-Control-Schicht, kurz MAC-Layer genannt, an. Die folgende Auswahl beschränkt sich auf energieeffiziente Protokolle für Ad-hoc-Netze und enthält keine Protokolle für Funknetze mit Infrastruktur.

IEEE 802.11 PSM Bereits der IEEE Standard 802.11 [IEEE 802.11] sieht einen Energiesparmodus für Funknetzkarten bei Betrieb in Ad-hoc-Netzen vor. Dieser Modus wird *Power Save Mode*, kurz PSM, genannt.

Alle Knoten des Netzes synchronisieren sich über einen verteilten Algorithmus gemeinsam und vereinbaren, dass sie regelmäßig alle gleichzeitig wach sind. Diese kurze, gemeinsame Wachphase wird auch ATIM[9]-Window genannt. Während ihr können alle Knoten das Senden von Nachrichten ankündigen. Die betroffenen Empfängerknoten wechseln nach der gemeinsamen Wachphase nun nicht in den Schlafmodus, sondern bleiben bis Ende des nächsten ATIM-Windows wach, um die für sie bestimmten Nachrichten entgegennehmen zu können. Ein gesamter Zyklus aus ATIM-Window und anschließendem Zeitraum für die Übertragung der Nutzdaten wird *Beacon-Intervall* genannt, weil zur Synchronisation der Knoten regelmäßig kleine Datenpakete, so genannte Beacons, versendet werden.

Verbesserungen am IEEE Standard 802.11 In [EunVai 2002] werden einige Modifikationen am Energiesparmodus des IEEE Standards 802.11 vorgenommen. Die beiden zentralen Änderungen sind:

1. Knoten, die innerhalb eines Beacon-Intervalls alle Nutzdaten empfangen und gesendet haben, können sich bis zum Beginn des nächsten ATIM-Windows wieder schlafenlegen.

2. Jeder Knoten passt die Länge des ATIM-Windows für seine Bedürfnisse an und teilt diesen Wert den anderen Knoten in jedem gesendeten Datenpaket mit.

Während die erste Änderung recht einfach zu implementieren ist und kaum weitere Folgen nach sich zieht, benötigt die zweite Änderung doch einige Modifikationen des Protokolls. Jeder Knoten muss nun seine Ankündigungen sortieren, so dass jene Knoten mit kurzen Wachphasen zuerst benachrichtigt werden. Außerdem wird eine Statistik über die Anzahl der Versuche, eine Nachricht anzukündigen, geführt. Dasselbe gilt für die Zahl der Nachrichten, die auf Grund gescheiterter Ankündigungen erst im nächsten oder übernächsten Beacon-Intervall empfangen werden konnten. Mit Hilfe solcher Statistiken kann jeder Knoten die Länge seines ATIM-Windows passend wählen.

Das Ergebnis ist ein Protokoll, das in jedem Fall höchstens ähnlich viel Energie benötigt wie IEEE 802.11 PSM. Im günstigsten Fall benötigt es nur ein Viertel der Energie.

[9] Announcement Traffic Indication Message

PAMAS In [Karn 1990] wird mit dem Protokoll *Multiple Access with Collision Avoidance*, kurz MACA, der Grundstein für das Reservieren des Mediums mit RTS und CTS nach IEEE 802.11 gelegt. Mit MACA konnte das Problem des *Hidden Terminals* einfach auf logischer Ebene gelöst werden. In [Karn 1990] wird auch das „Blockieren“ des Verstärkers von Knoten genannt, deren geographische Nachbarn gerade Daten empfangen.

Einen Schritt weiter, vom Blockieren zum energiesparenden Schlafmodus, geht das Protokoll PAMAS [SinRag 1998]. Allerdings verwendet PAMAS Außerbandsignalisierung, das heißt zwei verschiedene Kanäle für Kontrollnachrichten und Daten.

Durch zeitmultiplexen Betrieb könnte das Protokoll auch bei nur einem Kanal verwendet werden. Energieeffizienz und Durchsatz würden dabei aber etwas einbüßen. PAMAS ist dem Protokoll aus IEEE 802.11 in zwei Punkten überlegen:

1. Es sind keine festen Zyklen, wie die Beacon-Intervalle in IEEE 802.11, nötig.
2. Ein Knoten, der nach einer Schlafphase aufwacht und feststellt, dass ein benachbarter Knoten gerade eine Nachricht empfängt, kann über den Signalisierungskanal sofort erfragen, wie lange diese Übertragung noch dauert und sich entsprechend wieder schlafenlegen.

Im Falle eines Netzes, bei dem nur sporadisch Nachrichten ausgetauscht werden, spart PAMAS kaum Energie, da benachbarte Knoten nur während einer Nachrichtenübertragung in den Schlafmodus wechseln.

S-MAC und T-MAC In [WeHeEs 2002] wird S-MAC, ein energieeffizientes MAC-Verfahren für Sensornetze, vorgeschlagen. Da es einige Gemeinsamkeiten zwischen Sensornetzen und den hier betrachteten Netzen gibt, können Ideen für Sensornetze hier verwendet werden.

Anders als in PAMAS legt S-MAC einen Knoten auch dann regelmäßig schlafen, wenn gerade kein benachbarter Knoten eine Nachricht empfängt. Damit die Knoten trotz der periodischen Wechsel zwischen Schlaf- und Wachphasen miteinander kommunizieren können, synchronisieren sie sich untereinander, ähnlich wie nach IEEE 802.11. Ein paar wenige Knoten, die *Synchronizer*, geben den Zeitplan für die Wachphasen vor. Die anderen Knoten, genannt *Follower*, adaptieren den Zeitplan.

Dadurch bilden sich *Cluster* – Knotengruppen, die denselben Zeitplan verwenden. Knoten an der Grenze zweier benachbarter Cluster müssen allerdings öfters aufwachen, wenn sie mit Knoten aus beiden Clustern kommunizieren möchten. Sie adaptieren gegebenenfalls beide Zeitpläne.

Die Schlaf- und die Wachphasen werden lang gewählt. Dies erhöht zwar die Latenz, vereinfacht aber die Synchronisation, da die Uhrendrift gegenüber der Wachphase klein ist.

S-MAC verwendet RTS/CTS, um Nachrichtenübertragungen wie in IEEE 802.11 vorzubereiten. Jeder potentielle Sender wartet am Beginn der gemeinsamen Wachphase eine zufällig gewählte Zeit ab und sendet dann RTS. Lange Nachrichten werden in kurze Pakete zerteilt, die einzeln mit ACK bestätigt werden. Dadurch muss bei einer gestörten Übertragung nur das kurze Paket und nicht die ganze, lange Nachricht wiederholt werden.

Das Protokoll T-MAC [DamLan 2003] optimiert S-MAC und kommt in einigen Szenarien mit nur einem Fünftel der Energie aus. Während bei S-MAC Wach- und Schlafphase jeweils

feste Länge haben, passt T-MAC sie dynamisch an. Hat ein Knoten innerhalb eines gewissen Zeitintervalls nach Beginn der gemeinsamen Wachphase weder ein an ihn gerichtetes RTS, noch einen Broadcast empfangen und möchte er auch nichts senden, so wechselt er in die Schlafphase.

Damit wird aber ein großes Problem geschaffen. Ein potentieller Sender kann am Senden von RTS durch die Kommunikation zwischen zwei anderen Knoten gehindert sein. Sein potentieller Empfänger bekommt davon nichts mit und wechselt in die Schlafphase. Folglich müsste der potentielle Sender bis zum Beginn der nächsten gemeinsamen Wachphase warten. Dagegen bietet T-MAC zwei Möglichkeiten:

1. *FutureRTS:* Ein potentieller Sender, der durch die Kommunikation zwischen benachbarten Knoten blockiert wird, kann direkt nach dem Mithören von CTS schnell noch ein FutureRTS an seinen potentiellen Empfänger senden. Dieser weiß dadurch, nach welcher Zeit er sich für eine Nachricht wieder bereithalten muss, und wechselt nur kurzzeitig in den Schlafmodus.

2. *Full-Buffer-Priority:* Empfängt ein potentieller Sender ein an ihn gerichtetes RTS, bevor er selbst RTS gesendet hat, so kann er sofort senden, statt mit CTS zu antworten. Dieses Vorgehen wird nur für Knoten mit besonders vollen Sendepuffern empfohlen.

5.4 Routing

Da der Clusterhead in den hier diskutierten Verfahren zur energieeffizienten Dienstnutzung ähnlich einem Router arbeitet, werden im Folgenden einige energieeffiziente Routing-Algorithmen betrachtet.

In [SiWoRa 1998], einer frühen Arbeit zum energieeffizienten Routing in Ad-hoc-Netzen, werden fünf Anforderungen an ein entsprechendes Verfahren genannt:

1. *Minimiere den Energieverbrauch je gesendetem Paket:* Dieser Anforderung bei wird geringer Netzlast durch Shortest-Path-Routing genüge getan. Shortest-Path-Routing wird auch in vielen „normalen“ Routing-Verfahren verwendet.

   Wegen Kollisionsvermeidung und langer Wartezeiten wird bei hoher Netzlast der Weg über bestimmte Knoten aber so teuer, dass ein Umweg weniger Energie benötigt.

2. *Maximiere die Zeit bis zur Netzpartitionierung:* Eine Partitionierung des Netzes tritt ein, wenn alle Knoten an einem Engpass des Netzes ausfallen. Stellen, an denen wenige Knoten zwei oder mehr Teile der Netztopologie verbinden, sollten daher identifiziert und die entsprechenden Knoten nur abwechselnd verwendet werden.

3. *Belaste alle Knoten gleich:* Es darf keinen Knoten geben, dessen Energievorrat durch das Routing signifikant schneller als der anderer Knoten aufgebraucht wird.

4. *Ein Paket darf auf seiner Route nur Knoten mit möglichst großem Energievorrat passieren:* Diese Forderung schont Paket für Paket jene Knoten, die nur noch einen kleinen Energievorrat haben.

5. *Minimiere die maximalen Energiekosten aller Knoten, die beim Routing eines Pakets auftreten:* In [SiWoRa 1998, Kap. 3] wird sogleich eingestanden, dass diese Anforderung bei einem konkreten Algorithmus wohl nicht direkt erfüllt werden kann. Bemüht man sich aber, die Kosten je geroutetem Paket an jedem Knoten so weit wie möglich zu senken, so senkt man auch das Maximum dieser Kosten.

[SriChi 2002] teilt energieeffiziente Ad-hoc-Routing-Algorithmen in zwei Klassen ein:

1. *Distributed:* Alle Knoten im Netz übernehmen die gleichen Aufgaben beim Routing. Selbstständig entscheiden sie an Hand eigener Informationen oder von Nachbarknoten, ob sie zeitweise in einen Schlafmodus wechseln und die Funknetzschnittstelle abschalten können, oder ob sie als Router benötigt werden. BECA, AFECA und GAF gehören in diese Klasse.

2. *Backbone-based:* Einige wenige Knoten im Netz werden Koordinatoren und bestimmen, welche Knoten in ihrer Nachbarschaft wach bleiben müssen und welche Knoten sich schlafenlegen dürfen. Zu dieser Klasse gehören CEC und CPC.

   SPAN ist eine Ausnahme. Das Verfahren verwendet zwar einen Backbone, die Schlafzeiten der Knoten werden allerdings nicht von den Koordinatoren, sondern vom MAC-Layer verwaltet.

Allen folgenden Verfahren ist gemein, dass sie zwar energieeffizientes Routing ermöglichen, dabei aber nicht das eigentliche Routing-Protokoll festlegen. Meistens wurden sie mit Ad hoc On Demand Distance Vector Routing [PeBeDa 2003], kurz ADOV, getestet.

BECA und AFECA Bei BECA [YaHeEs 2000], dem Basic Energy Conserving Algorithm, nimmt zunächst jeder Knoten am Routing teil. Hat ein Knoten eine gewisse Zeit lang weder eine Nachricht gesendet oder empfangen noch geroutet, so wechselt er in einen Schlafzustand und schaltet die Netzwerkschnittstelle ab. Der Knoten wacht erst nach Ablauf einer festen Zeit $T_S = T_0$ oder wenn er selber eine Nachricht senden möchte wieder auf und beteiligt sich am Routing. Auf jede Nachricht wird bei BECA mit einer Empfangsbestätigung geantwortet. Bleibt diese aus, so geht der Sender der Nachricht davon aus, dass der Empfänger gerade schläft. Er wiederholt die Nachricht später.

Der Adaptive Fidelity Energy Conserving Algorithm, kurz AFECA, ist eine Variante von BECA, bei der T_S variabel gewählt wird. In einen Bereich des Netzes, in dem die Knoten spärlich verteilt sind, werden anteilig mehr Router benötigt, als in dichten Bereichen. In Letzteren kann ein Knoten bei gleicher Netzübertragungskapazität länger schlafen. Bei AFECA schätzt jeder Knoten die Zahl seiner Nachbarknoten durch Mithören von Nachrichten ab. Aus der Zahl der Nachbarn N bestimmt ein Knoten dann in der Wachphase die Dauer seiner nächsten Schlafphase zu $T_S = M \cdot T_0$, wobei M eine Zufallszahl zwischen 1 und N ist. Für $N = 1$ verhält sich ein Knoten bei AFECA und BECA gleich.

GAF GAF [YaHeEs 2001], der Geographical Adaptive Fidelity-Algorithmus, wurde von den Autoren von BECA und AFECA entwickelt. Allerdings verwendet er explizite Informationen

über die geographische Position und Bewegung jedes Knotens. Diese Informationen erhält GAF zum Beispiel von einem GPS[10]-Gerät.

GAF legt ein gleichmäßiges Rastergitter über das Ad-hoc-Netz, so dass sich jeder Knoten an Hand seiner Position einem Planquadrat des Gitters zuordnen kann. Die Planquadrate sind so gewählt, dass jeder Knoten eines Planquadrats mit jedem Knoten der vier benachbarten Planquadrate kommunizieren kann. Entsprechend reicht es aus, wenn ein Knoten je Planquadrat das Routing übernimmt. Jeder Knoten kennt in GAF drei Zustände:

- *Discovery:* Aktive Suche mit Broadcast-Nachrichten nach Knoten, die im selben Planquadrat liegen.
- *Active:* Der Knoten wird zum Router des Planquadrats.
- *Sleeping:* Wechsel in einen energiesparenden Schlafmodus, während ein anderer Knoten des Planquadrats das Routing übernimmt.

Welcher Knoten dabei das Routing übernimmt, hängt von der noch zu erwartenden Lebensdauer der einzelnen Knoten ab. Diese Zeit errechnet sich aus dem noch vorhandenen Energievorrat. Auf diese Weise werden die Energievorräte der Knoten eines Planquadrats gleichmäßig aufgebraucht.

Die Dauer der Schlafphase hängt von der noch zu erwartenden Lebensdauer – Energievorrat zu Leistungsaufnahme – des Knotens ab, der das Routing übernimmt.

GAF-ma ist eine spezielle Variante von GAF für hohe Knotenmobilität. Bewegen sich die Knoten sehr schnell, so erhöht sich die Wahrscheinlichkeit, dass ein Knoten, der als Router arbeitet, sein Planquadrat verlässt, lange bevor die anderen Knoten aufwachen und einen neuen Router wählen können. Daher bezieht GAF-ma bei der Berechnung der Schlafzeiten auch die Zeit ein, nach der der Router bei gleich bleibender Geschwindigkeit das Planquadrat verlassen wird.

CEC Der Cluster based Energy Conservation-Algorithmus [YaBi+ 2003], kurz CEC, basiert auf Ideen von GAF. Allerdings verwendet er keine expliziten Positionsinformationen. Statt der Zuordnung in Rastergitter bilden die Knoten Cluster.

Cluster werden durch das Versenden und Beantworten von Broadcast-Nachrichten ermittelt. Ein Cluster ist dadurch gekennzeichnet, dass ein Knoten in der Mitte des Clusters, der Clusterhead, direkt mit jedem anderen Knoten des Clusters kommunizieren kann. Zwei beliebige Knoten eines Cluster können sich daher in maximal zwei Hops erreichen. Die Cluster wiederum werden durch Gateways miteinander verbunden. Dabei werden zwei Arten von Gateways unterschieden:

- *Primäre* Gateways sind Knoten, die direkt mit zwei oder mehr Clusterheads kommunizieren können. Sie verbinden zwei Clusterheads über zwei Hops.
- *Sekundäre* Gateways sind Knoten, die ein anderes Gateway erreichen. Entsprechend verbinden sie zwei Clusterheads über drei Hops.

[10] Global Positioning System

Oft kommen für die verschiedenen Aufgaben mehrere Knoten in Frage. GAF entscheidet in diesem Fall an Hand der noch zu erwartenden Lebensdauer der einzelnen Knoten, wer welche Aufgabe übernimmt und wer sich schlafenlegen kann.

Bei der Wahl der Gateways werden Primäre Gateways bevorzugt und zunächst die Zahl der Cluster berücksichtigt, die ein Knoten verbindet, bevor die Energievorräte verglichen werden.

Wie bei GAF, wachen bei CEC diejenigen Knoten eines Clusters, welche nicht als Router arbeiten, nach einer gewissen Zeit für die Neuwahl von Clusterhead und Gateways wieder auf.

In [YaBi+ 2003, Kap. 4.3] werden auch Vorschläge gemacht, wie mit einer Nachricht umgegangen werden kann, die für einen Knoten bestimmt sind, der gerade schläft:

- Falls die Nachricht von mehreren Knoten verarbeitet werden kann, nimmt sie ein Knoten entgegen, der gerade wach ist.
- Der Clusterhead puffert die Nachricht, bis der Empfänger aufwacht.

Bei hoher Knotenmobilität kann auch bei CEC zur Berechnung der Schlafdauer die Zeit miteinbezogen werden, nach der der Clusterhead die Mitte des Clusters verlassen hat und nicht mehr als solcher arbeiten kann. Dazu muss allerdings die Geschwindigkeit der Knoten, oder zumindest eine Oberschranke, vorab bekannt sein.

SPAN Der verteilte Algorithmus SPAN [BeJa+ 2001] bildet einen *Backbone*, über den Nachrichten zu weiter entfernten Knoten geroutet werden. Die Knoten des Backbones werden auch *Koordinatoren* genannt. Ein Knoten, der nicht Koordinator ist, kann sich nach dem Power Save Mode von IEEE 802.11 schlafenlegen.

SPAN stellt sicher, dass eine geeignete Anzahl von Knoten die Aufgabe eines Koordinators wahrnehmen. Ein Knoten kann prinzipiell nur dann Koordinator werden, wenn er als solcher zwei Knoten in seiner Nachbarschaft verbindet, die sonst gar nicht, oder nur über mehr als zwei Koordinatoren miteinander kommunizieren können. Die entsprechenden Topologiedaten erhält jeder Knoten durch regelmäßige Broadcasts benachbarter Knoten. Jeder Knoten prüft regelmäßig, ob er für die Funktion eines Koordinators in Frage kommt. Falls ja, so wartet er zunächst eine gewisse Zeit, abhängig von dem noch zur Verfügung stehenden Energievorrat und von der Anzahl der Knotenpaare, die er verbinden würde. Hat nach dieser Wartephase zwischenzeitlich kein benachbarter Knoten die Aufgabe übernommen, so wird er anschließend selbst Koordinator.

Um alle Knoten gleichmäßig zu belasten, sollte die Funktion eines Koordinators regelmäßig abgegeben werden. Dazu kann sich ein Koordinator als „provisorisch“ markieren, so dass ihn seine Nachbarn als Koordinator verwenden können, bei der Entscheidung selbst Koordinator zu werden aber als normalen Knoten betrachten.

SPAN kann ohne Modifikationen direkt mit dem MAC-Layer nach IEEE 802.11 und dessen Power Save Mode [IEEE 802.11, Kap. 11.2.2] verwendet werden. Allerdings schlagen die Erfinder von SPAN drei Modifikationen vor, die die Energieeffizienz weiter erhöhen [BeJa+ 2001, Kap. 4.4] sollen:

1. Koordinatoren sind immer wach. Sie brauchen die untereinander gesendeten Nachrichten nicht im ATIM-Window anzukündigen.

2. Nach IEEE 802.11 PSM muss ein Knoten, der mehrere Broadcast-Nachrichten innerhalb eines Beacon-Intervalls senden will, nur eine davon ankündigen, da anschließend alle Knoten während des ganzen Beacon-Intervalls wach bleiben. Da bei SPAN viele Broadcast-Nachrichten anfallen, könnten die Knoten nur selten in den Schlafmodus wechseln.

   Wird *jede* Broadcast-Nachricht im ATIM-Window angekündigt, so kennt jeder Knoten die Zahl der Broadcast-Nachrichten vorab und kann sich nach deren Empfang bis zum nächsten ATIM-Window schlafenlegen.

3. Die Zeit des Beacon-Intervalls wird nach dem ATIM-Window in zwei Intervalle geteilt. Im ersten Intervall, dem *Advertised Traffic Window*, werden alle Nachrichten von oder an Knoten gesendet, die nicht Koordinatoren sind. Während dem zweiten Intervall können sich diese Knoten schlafenlegen und die Koordinatoren miteinander kommunizieren.

Verbesserungen an SPAN In [IkiOga 2003] werden zwei Modifikationen an SPAN vorgeschlagen, die den Energieverbrauch weiter reduzieren sollen.

1. Das Advertised Traffic Window[11] aus der Modifikation von IEEE 802.11 PSM wird variabel gewählt. Je weniger Nachrichten von oder zu Knoten, die nicht Koordinatoren sind, gesendet werden, desto kürzer wird es.

2. Ein Knoten, der nicht Koordinator ist und nach zwei Beacon-Intervallen keine Nachricht erhalten hat, wacht fortan nur noch in jedem zweiten ATIM-Window auf. Die Koordinatoren stellen sich darauf entsprechend ein, indem sie nach einer gescheiterten Ankündigung nicht gleich davon ausgehen, dass der entsprechende Empfängerknoten den Cluster verlassen hat, sondern einen weiteren Versuch unternehmen.

Durch einen neuen Algorithmus für die Wahl der Koordinatoren soll die Lebensdauer des gesamten Netzes erhöht werden. Im Gegensatz zu SPAN werden die Energievorräte der verschiedenen Knoten explizit verglichen und bei Bedarf eine Neuwahl des Koordinators veranlasst.

CPC Der Coordinated Power Conservation-Algorithmus [SriChi 2002] ähnelt dem Algorithmus CEC [YaBi+ 2003]. Die Menge aller Knoten des Netzes wird in Dominating Sets, gleich den Clustern bei CEC, geteilt. Diese bestimmen je einen Knoten, den Virtual-Access-Point (VAP), der von allen Knoten des Dominating Sets in einem Hop erreicht werden kann. Die verschiedenen VAPs werden in CPC über zwei oder drei Hops zur Bildung eines Backbones miteinander verbunden. Dazu dienen ausgezeichnete Knoten, die den Gateways in CEC entsprechen.

Im Routing unterscheiden sich CEC und CPC stark, da am Routing nicht nur die VAPs und Gateways, sondern alle Knoten des Netzes teilnehmen. Die Autoren von [SriChi 2002]

[11] Das Advertised Traffic Window wird in [IkiOga 2003] New Advertisement Window (NATIM) genannt. Der Begriff ist unsauber gewählt, da in diesem Window keine Ankündigungen, sondern die eigentlichen Nachrichten versendet werden. Dasselbe gilt für die Abkürzung NATIM, offensichtlich von ATIM (Announcement Traffic Indication Message [IEEE 802.11, Kap. 4]) abgeleitet.

nennen dieses Verfahren *Backbone-coordinated* im Gegensatz zu *Backbone-forwarded* bei CEC. Die VAPs brauchen nur die Schlafzeiten der Knoten ihres Dominating Sets zu verwalten und müssen nur selten als Router arbeiten. Das Verfahren hat zwei Vorteile:

1. Die Knoten des Backbones werden entlastet und der Energieverbrauch verteilt sich gleichmäßiger auf alle Knoten.

2. Die Bandbreite des Netzes wird nicht durch die Knoten des Backbones begrenzt, sondern kann durch Hinzunahme oder Schlafenlegen von Knoten beliebig eingestellt werden.

Ein Nachteil ist, dass die Knoten außerhalb des Backbones nur kurze Schlafzeiten haben dürfen, um gegebenenfalls rasch als Router einsetzbar zu sein.

5.5 Diensterkennung

Zwar existieren heute eine Reihe von Verfahren zur Diensterkennung in Ad-hoc-Netzen, doch wird Energieeffizienz nur in sehr wenigen berücksichtigt.

Einige gängige Verfahren zur Diensterkennung werden in [Angstm 2003] beschrieben. Besprochen wird dort unter anderem Salution, Universal Plug and Play (UPnP), das Bluetooth Service Discovery Protocol (SDP), JINI von der Firma Sun Microsystems und der Intentional Naming Service (INS). Leider kann keines dieser Protokolle als energieeffizient bezeichnet werden.

Diensterkennung in DEAPspace DEAPspace ist ein Projekt der IBM Zürich, das energieeffiziente Diensterkennung und -nutzung in Single-Hop-Ad-hoc-Netzen thematisiert. In [Nidd 2001] wird ein Algorithmus zur energieeffizienten Diensterkennung aus diesem Projekt präsentiert.

Bei der Diensterkennung in DEAPspace haben alle Knoten die gleiche Aufgabe – es gibt keinen Clusterhead oder Koordinator. Ein Knoten sucht nicht aktiv nach Diensten, sondern bekommt alle benötigten Informationen ohne Aufforderung zugestellt.

Eine Dienstbeschreibung enthält bei DEAPspace nicht nur den Namen des Dienstes und die Kennung des zugehörigen Dienstknotens, sondern auch eine Verfallszeit. Jeder Knoten sendet regelmäßig alle ihm bekannten Dienstbeschreibungen im Broadcast an alle anderen Knoten. Dies beinhaltet nicht nur die Beschreibungen seiner lokalen Dienste, sondern auch alle Dienstbeschreibungen, die er von anderen Knoten erhalten hat.

Empfängt ein Knoten einen solchen Broadcast, aktualisiert er seine Sicht auf die Gesamtsituation entsprechend.

Die Gesamtzahl der Broadcasts ist unabhängig von der Zahl der beteiligten Knoten gewählt. Je mehr Knoten an dem Algorithmus teilnehmen, desto seltener sendet der einzelne Knoten seine Sicht aller Dienste.

Damit die Dienstbeschreibungen auch bei vielen Knoten nicht verfallen, können Knoten, die einen Dienst besitzen, dessen Beschreibung bald abläuft, kurzfristig ihre Sicht aller Dienste mit einer aktualisierten Beschreibung der eigenen Dienste senden.

In [Nidd 2001] wird vorgerechnet, dass auch bei relativ vielen Knoten die Diensterkennung so nie mehr als 5% der Bandbreite des gesamten Netzes benötigt.

In DEAPspace werden die Dienste neuer Knoten rascher von anderen Knoten erkannt als bei Diensterkennungsverfahren, in denen jeder Knoten nur die Beschreibungen der eigenen Dienste in einem Broadcast mitteilt. Dank der regelmäßigen Broadcasts können die Knoten in einen Schlafmodus wechseln, ohne die Sicht auf die Dienste dauerhaft zu verlieren. Da nicht zu viele Knoten diesen energieeffizienten Modus verwenden sollten, wird er nur für Knoten mit sehr geringem Energievorrat vorgeschlagen.

Lanes kontra Flooding In [KlKoOb 2003] werden Nachteile der Diensterkennung durch Flooding, wie es in großen Ad-hoc-Netzen oft zum Einsatz kommt, diskutiert. Beim Flooding – zu Deutsch Fluten des Netzes – leitet jeder Knoten eine eben erhaltene Diensterkennungsanfrage an alle in Reichweite befindlichen Knoten weiter. Erhält er die Anfrage ein weiteres Mal, so ignoriert er sie. Dieses Vorgehen stellt sicher, dass jeder Knoten die Anfrage so schnell wie möglich erhält. Allerdings entstehen sehr viele unnötige Nachrichten.

In [KlKoOb 2003] wird der Vorschlag gemacht, so genannte Lanes für die Verbreitung von Dienstbeschreibung und für die Beantwortung von Diensterkennungsanfragen zu verwenden. Dazu wird eine logische, zweidimensionale Struktur, wie ein Koordinatensystem, über das Netz gelegt. Das gesamte Netz wird durch ein großes Rechteck $(0,0) \times (D_x, D_y)$ repräsentiert. Dieses Rechteck wird in Spalten $(x_i, 0) \times (x_{i+1}, D_y)$, die so genannten Lanes, zerlegt. Jeder Knoten des Netzes repräsentiert nun einen Abschnitt einer solchen Lane. Er kennt den Knoten des Abschnittes über ihm und den Knoten des Abschnittes unter ihm. Außerdem kennt jeder Knoten eine Anycast-Adresse, die alle Knoten der Lane zu seiner Linken adressiert. Entsprechendes gilt für alle Knoten der Lane zu seiner Rechten.

Jeder Knoten teilt seine Dienstbeschreibungen allen Knoten seiner Lane mit. Dies erfolgt durch sequentielles Weiterleiten der Dienstbeschreibung in y-Richtung. Diensterkennungsanfragen laufen hingegen in x-Richtung von Lane zu Lane, wie in Abbildung 8 gezeigt. Je Lane wird nur ein Knoten befragt, da er alle Dienste von allen Knoten der Lane kennt.

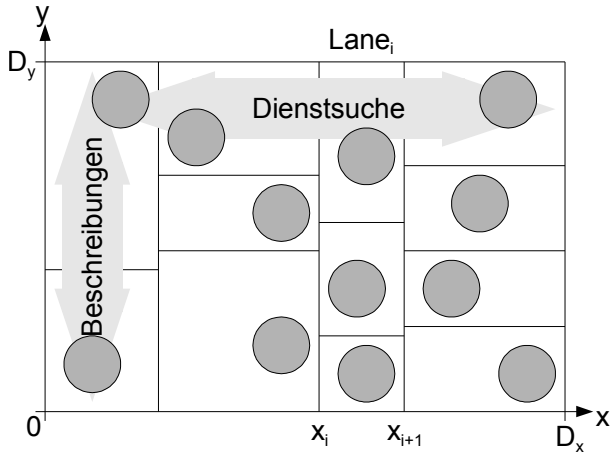

Abbildung 8: Lanes zur Diensterkennung

Auf diese Weise können Diensterkennungsanfragen ohne großen Nachrichtenüberschuss beantwortet werden. Dieses Vorgehen ist in jedem Fall effizient und spart verglichen mit Flooding auch Energie, ist aber im Sinne dieser Arbeit kein energieeffizienter Algorithmus, da Schlafphasen nicht explizit ermöglicht werden. Im Gegenteil: Durch das sequentielle Weiterleiten von Diensterkennungsanfragen würden sich Verzögerungen durch Schlafphasen unangenehm aufsummieren.

5.6 Dienstnutzung bei Kenntnis der Anwendung

Die in [StGa+ 1996] formulierte Empfehlung, durch regelmäßiges Abschalten der Funknetzwerkschnittstelle Energie zu sparen, wird dort exemplarisch auf Email und Browser angewendet. Zum Empfang von Emails wird die Funknetzwerkkarte nur alle paar Minuten kurz aktiviert und die Emails abgerufen. Bei einer Browser-Anwendung wird die Schnittstelle aktiviert, sobald der Benutzer eine neue Seite anwählt. Anschließend wird die Schnittstelle wieder in einen Schlafmodus versetzt, sobald alle Daten übertragen wurden und der Benutzer eine gewisse Zeit keine neue Seite angewählt hat.

Diese einfachen Beispiele verdeutlichen, dass bei exakter Kenntnis der Anwendung und durch spezialisierte Protokolle bei den Netzwerkschnittstellen sehr viel Energie eingespart werden kann. Im Folgenden werden ähnliche anwendungsspezifische Protokolle betrachtet.

Videodatenströme Die Datenströme typischer Multimedia-Anwendungen bestehen aus einer dichten Folge von Datenpaketen. In [Chandra 2002] werden die Datenströme der Videoformate Microsoft Windows Media, Real Media und Apple Quick Time untersucht. Besonderes Augenmerk liegt auf den Zwischenankunftszeiten der einzelnen Pakete.

Dabei zeigt sich, dass die Zwischenankunftszeiten eine geringe Varianz haben und sich daher relativ gut vorhersagen lassen. Windows Media erweist sich diesbezüglich als besonders stabil.

Diese Erkenntnis verwenden die Autoren von [Chandra 2002] für einen einfachen, klientenseitigen Energiesparmodus: Die Netzwerkschnittstelle misst ständig die Zwischenankunftszeiten und berechnet das Mittel der letzten N Messungen. Dann kann sie sich nach dem Empfang eines Datenpakets für eben diese Zwischenankunftszeit abzüglich einer kleinen Toleranz schlafenlegen.

Je nach Videoformat und Toleranz kann so auch bei relativ hoher Übertragungsrate (768 Kbps) und nur geringen Paketverlusten ($\approx 0{,}25\%$) der Energieverbrauch mehr als halbiert werden.

In einer weiterführenden Arbeit [ChaVah 2002] wird versucht, den Power Save Mode von IEEE 802.11 [IEEE 802.11, Kap. 11.2] zur energieeffizienten Übertragung von Videodatenströmen zu verwenden. Dieser erweist sich als nur bedingt geeignet, da IEEE 802.11 PSM von sporadisch ankommenden Datenpaketen ausgeht und existierende Implementierungen die Beacon-Intervalle nicht an die Zwischenankunftszeiten der Pakete in den Datenströmen anpassen.

Anschließend wird eine Lösung präsentiert, bei der der Multimedia-Server die Klienten über die Zwischenankunftszeiten der Datenpakete informiert und sie entsprechend gleichmäßig

überträgt. Dadurch ist eine noch energieeffizientere Datenübertragung bei noch geringeren Paketverlusten möglich, als bei einer rein klientseitigen Anpassung.

Anwendungsschnittstelle zum Energiesparmodus [KraKri 2000] beschreibt ein Verfahren, das mit der Verwendung von Anwendungswissen eine energieeffiziente Nachrichtenübertragung zwischen einer Basisstation und einem mobilen Knoten ermöglicht. Der mobile Knoten gibt vor, wann er schläft und wann er wach ist. Während der Schlafphase puffert die Basisstation alle Nachrichten, die für ihn bestimmt sind.

Die Entwickler testen ihr Verfahren an drei typische Anwendungen: Email, Websurfen und das Verfassen von Dokumenten im Team. Sie identifizieren die unterschiedlichen *Kommunikationsmuster* der Anwendungen nach Anzahl und Zwischenankunftszeiten der Nachrichten und dem Initiator der Kommunikation. Je nach Anwendung lässt sich das Kommunikationsmuster genau vorhersagen und entsprechend bei der Nachrichtenübertragung ohne größere Latenzen Energie sparen.

In [KraKri 2000, Kap. 5] wird vorgeschlagen, den Anwendungen eine Schnittstelle zu bieten, mit der sie Einfluss auf die Strategien nehmen können, nach denen das Energiesparverfahren Schlaf- und Wachphasen ermittelt. Damit soll das Verfahren optimal an das Kommunikationsmuster einer Anwendung angepasst werden können.

6 Entwicklung eines konkreten Verfahrens: LATE RISER

Nach der theoretischen Diskussion verschiedener Verfahren zur energieeffizienten Nutzung von Diensten wurde entsprechend den Vorgaben ein konkretes Verfahren aus SANDMAN [ScBeRo 2004] entwickelt. Es wurde *LATE RISER* – zu Deutsch „Langschläfer“ – getauft, da die Schlafphasen deutlich länger sind, als bei gängigen energieeffizienten Algorithmen [SinRag 1998, IEEE 802.11, WeHeEs 2002]. Viele Erkenntnisse aus den in Kapitel 5 betrachteten Arbeiten flossen bei der Entwicklung von LATE RISER mit ein.

Da das Verfahren im Rahmen dieser Arbeit auch implementiert wurde, musste es aus zeitlichen Gründen einfach gehalten werden. Viele der in Kapitel 4.3 genannten Nachteile konnten nicht durch gesonderte Behandlung ausgeglichen werden.

Bevor die verschiedenen Funktionen und Aspekte von LATE RISER im Folgenden detailliert beschrieben werden, ist in Abbildung 9 eine Dienstregistrierung, -erkennung und -nutzung in LATE RISER exemplarisch dargestellt.

6.1 Knoten und Cluster

LATE RISER verwendet Clusterbildung wie SANDMAN. Jeder Knoten wird genau einem Cluster zugeordnet. Es werden zwei Arten von Knoten unterschieden:

- *Clusterheads:* Ein Knoten, der Clusterhead ist, koordiniert einen Cluster, wie in Kapitel 2.4 erläutert. Jeder Cluster hat genau einen Clusterhead. Dieser kann der einzige Knoten des Clusters sein. Ein Clusterhead ist immer wach.

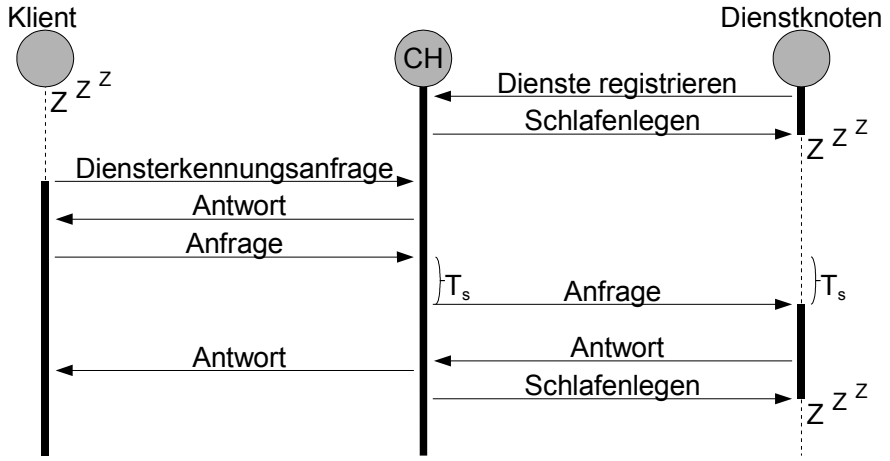

Abbildung 9: Dienstregistrierung, -erkennung und -nutzung in LATE RISER

- *Clusterknoten:* Ein Knoten, die nicht Clusterhead ist, wird als Clusterknoten bezeichnet. Clusterknoten existieren nur in Clustern, die aus zwei oder mehr Knoten bestehen. Ein Clusterknoten kann durch „seinen“ Clusterhead – den Clusterhead seines Clusters – in den Schlafmodus versetzt werden.

Für die Clusterbildung in LATE RISER kann ein beliebiger Algorithmus mit einem beliebigen Clustermanagement verwendet werden. Voraussetzung ist lediglich, dass ein Clusterhead seine Clusterknoten, wie bei SANDMAN, in Single-Hop-Verbindungen erreichen kann. Geeignete Algorithmen sind zum Beispiel in CPC [SriChi 2002] oder CEC [YaBi+ 2003] enthalten, die in Kapitel 5.4 erläutert werden.

LATE RISER benötigt vom Clustermanagement nur wenig Informationen: Jeder Knoten muss erfragen können, wer sein Clusterhead ist und über einen Wechsel desselben informiert werden. Ein Clusterhead muss alle Clusterknoten seines Clusters ermitteln können und über Änderungen an der Clusterkonfiguration – die Menge der Knoten, die den Cluster bildet – informiert werden. Änderungen an der Clusterkonfiguration ergeben sich, wenn Knoten den Cluster verlassen oder ihm beitreten.

6.2 Erkennen anderer Knoten

In einem Ad-hoc-Netz kennen sich die Knoten in der Regel vorab nicht. Entsprechend wird ein Algorithmus zur Erkennung anderer Knoten, im Englischen *Discovery* genannt, benötigt.

Dieser wird hier nicht genauer festgelegt, da LATE RISER von diesem lediglich zwei Informationen benötigt: Zum Ersten die Knotenkennungen der in Reichweite befindlichen Knoten – beispielsweise in Form von IP-Adressen – und zum Zweiten muss für jeden Knoten ersichtlich sein, ob es sich um einen Clusterhead oder um einen Clusterknoten handelt.

Es reicht sogar aus, wenn sich Clusterknoten nur ihrem Clusterhead zu erkennen geben. Clusterheads hingegen müssen sich allen Knoten zu erkennen geben, da bei LATE RISER

jeder Knoten *jeden* Clusterhead zur Diensterkennung und -nutzung verwenden kann.

6.3 Diensterkennung

Wie bei SANDMAN diskutiert, kann clusterbasierte Diensterkennung auf einfache Weise mit einem Dienstverzeichnis realisiert werden. Die in [Angstm 2003] und Kapitel 5.5 diskutierten Verfahren zeigen, dass mit Clustern und Dienstverzeichnissen sehr energieeffiziente Diensterkennung möglich ist. LATE RISER übernimmt dieses Vorgehen. Ein Clusterknoten sendet seinem Clusterhead eine Beschreibung seiner Dienste zu, sobald sich ein Dienst geändert hat oder der Clusterhead neu gewählt wurde.

Wie oben genannt, kann ein Knoten in der Rolle des Klienten aber nicht nur Dienste des eigenen Clusters erkennen, sondern beliebige Clusterheads befragen. Ein Klient sendet seine Anfragen aber immer an einen Clusterhead und erhält von diesem auch die Antwort.

Bei LATE RISER legt der Clusterhead für jeden Dienst, der in seinem Cluster vertreten ist, einen Stellvertreterdienst an. Für gleiche Dienste – so genannte *Dienstklassen* – wird ein gemeinsamer Stellvertreterdienst verwendet. Gleichheit von Diensten wird bei LATE RISER an Hand der Dienstbeschreibungen erkannt, mit denen Knoten ihre Dienste beim Clusterhead anmelden.

Um im Folgenden zwischen Diensten, Dienstklassen und den eigentlichen Diensten sprachlich klar zu trennen, werden die eigentlichen Dienste – real existierende und entfernt aufrufbare Programminstanzen – fortan *Dienstinstanzen* genannt.

Auf Diensterkennungsanfragen hin teilt der Clusterhead immer die Beschreibung eines Stellvertreterdienstes mit. Die Dienstinstanzen werden dadurch, wie in der Aufgabenstellung gefordert und in Kapitel 3.4 beschrieben, vollständig vor den Klienten versteckt.

6.4 Zustandslose und zustandsbehaftete Dienste

Wie in Kapitel 2.5 diskutiert können sich Dienste in Schnittstellen und Nutzung stark unterscheiden.

Bei der Kooperationsform der Dienste geht LATE RISER von *einem* Klientknoten und *einem* Dienstknoten aus. Zusammengesetzte Dienste werden daher nur unterstützt, wenn sie dem klassischen Client-Server-Modell entsprechen.

Weiter setzt LATE RISER voraus, dass die Kommunikation zwischen Klient und Dienst dem Anfrage-Antwort-Muster folgt. Die Antwort kann auch wegfallen, wenn dies aus der Dienstbeschreibung vorab ersichtlich ist. Nicht unterstützt werden Dienste, wie Newsticker oder Event-Channel, die dem Klienten ohne unmittelbare Aufforderung Nachrichten zusenden.

Vorausgesetzt wird auch, dass Anfrage und Antwort als Ganzes geliefert werden. Strombasierte Datendienste, wie zum Beispiel Video, können nicht mit LATE RISER verwendet werden.

LATE RISER schränkt die Zeitkopplung bei der Dienstnutzung nicht ein. Dienste können synchron oder asynchron genutzt werden. Weiter werden zwei Arten von Diensten unterschieden:

- *Zustandslose Dienste:* Eine Folge von Anfragen eines Klienten muss nicht von derselben Dienstinstanz, sondern kann Anfrage für Anfrage von anderen Dienstinstanzen der gleichen Dienstklasse beantwortet werden.

  Für zustandslose Dienste braucht der Clusterhead die Informationen über eine Klient-Dienst-Beziehung nur vom Empfang der Anfrage bis zum Senden der Antwort – falls eine solche vorgesehen ist – zu speichern.

  Ein Beispiel für einen zustandslosen Dienst ist ein Sensor, aus dem in Kapitel 4.4 beschriebenen Pflanzenpflegesystem, der auf eine Anfrage hin die momentane Luftfeuchtigkeit misst und den Wert in der Antwort mitteilt.

- *Zustandsbehaftete Dienste:* Die Dienstnutzung wird implizit mit der ersten Anfrage durch den Klienten bei Clusterhead und Dienstknoten angemeldet. Der Clusterhead hält explizite Informationen über die Klient-Dienst-Beziehung und leitet zugehörige Anfragen immer an denselben Dienstknoten weiter.

  Intern verwendet LATE RISER für jede Klient-Dienst-Beziehung eine eigene Session, so dass Dienst und Clusterhead die verschiedenen Nachrichten korrekt zuordnen können.

  Anders aber als bei gängigen Sessionkonzepten, wie TCP, kann der Dienstknoten trotzdem zwischendurch in lange Schlafphasen wechseln. Das Sessionkonzept von LATE RISER garantiert eine dauerhafte eineindeutige Zuordnung, aber keine besonders schnellen Antwortzeiten.

  Ein Beispiel für einen zustandsbehafteten Dienst ist der in Kapitel 4.3 beschriebene Dateidienst, beim dem die Operationen Öffnen, Lesen und Schließen als separate Dienstnutzungen durchgeführt werden.

Dem Clusterhead muss explizit mitgeteilt werden, ob ein Dienst zustandslos oder zustandsbehaftet ist.

6.5 Koordinierung der Anfragen und Schlafphasen

Die Nutzung eines Dienstes erfolgt bei LATE RISER immer über dessen Stellvertreterdienst beim Clusterhead. Dieser kennt alle Clusterknoten, die entsprechende Dienstinstanzen bieten.

Auch wenn der Clusterhead selber einen Dienst bietet, wird trotzdem der Stellvertreterdienst verwendet. Dieses strikte Vorgehen garantiert, dass neue Dienstinstanzen leicht hinzugefügt werden können.

Verteilung der Anfragen Je Dienstklasse werden alle Anfragen vom entsprechenden Stellvertreterdienst zunächst in einer Warteschlange gepuffert und mit einem Zeitstempel versehen. Die anschließende Verteilung der Anfragen auf die in Frage kommenden Knoten nimmt ein Algorithmus $A_{Coordinator}$ vor. LATE RISER legt $A_{Coordinator}$ nicht fest. Der Algorithmus kann bei Bedarf an spezielle Dienstklassen angepasst werden.

Koordinierung der Schlafphasen Bearbeitet ein Clusterknoten keine Anfragen, so kann er in die Schlafphase versetzt werden. Dies ist ebenfalls Aufgabe von $A_{Coordinator}$.

Er bestimmt wann und wie lange ein Clusterknoten schlafen darf. Auch hierin wird $A_{Coordinator}$ von LATE RISER nicht festgelegt. Der Algorithmus kann wieder an spezielle Dienstklassen angepasst werden.

Bei LATE RISER besteht die Möglichkeit, dass ein Clusterknoten entgegen der Aufforderung durch seinen Clusterhead nicht in die Schlafphase wechseln kann – zum Beispiel wenn er als Klient selber einen Dienst nutzt. Auf eine Rückmeldung des betroffenen Clusterknotens wird der Einfachheit halber verzichtet. Stattdessen nimmt der Clusterhead an, der Knoten wäre eingeschlafen. Auch das vorzeitige Aufwachen eines Clusterknotens wird nicht berücksichtigt. Erst nach Ablauf der festgelegten Schlafdauer betrachtet auch der Clusterhead den Clusterknoten für wach.

$A_{Coordinator}$ sendet auch jene Knoten schlafen, die überhaupt keine Dienste bieten. Die Schlafphasen eines entsprechenden Clusterknotens können sehr, aber nicht beliebig lang sein. Möchte ein solcher Knoten nach einiger Zeit doch einen Dienst anbieten, so kann $A_{Coordinator}$ ihm nur nach Ende einer Schlafphase Anfragen zuteilen.

Verzahnung der Aufgaben In diesem Sinne übernimmt $A_{Coordinator}$ zwei Aufgaben: Die Verteilung von Anfragen *und* die Koordinierung der Schlafphasen. Da beide eng voneinander abhängen, ist eine Auftrennung in zwei Algorithmen nicht sinnvoll. Beispiele für diese Verzahnung sind:

- $A_{Coordinator}$ kann eine Anfrage zurückhalten und einem bald aufwachenden Knoten zuteilen, um einen Clusterknoten, der schon lange wach ist, in die Schlafphase versetzen zu können.
- $A_{Coordinator}$ kann einen Clusterknoten bewusst wach halten, auch wenn im Moment keine Anfragen vorliegen oder bearbeitet werden.

Aufruf des Algorithmus' Eine Neuberechnung nach $A_{Coordinator}$ ist notwendig, sobald sich ein Aspekt der möglichen Dienstnutzung ändert. Dazu gehören die Clusterkonfiguration, die angebotenen Dienste, die gepufferten Anfragen beim Clusterhead, ausstehende Antworten und die Schlafphasen der Clusterknoten.

LATE RISER deckt diese Aspekte ab, indem es $A_{Coordinator}$ nach folgenden Ereignissen aufruft:

- *Clusterknoten hinzugefügt oder entfernt:* Kommt ein Knoten zum Cluster hinzu, so sorgt der sofortige Aufruf von $A_{Coordinator}$ dafür, dass der Knoten rasch in die Schlafphase wechseln kann oder ihm eine Anfrage zugeteilt wird.

  Verlässt ein Knoten den Cluster so wird $A_{Coordinator}$ aufgerufen, um Anfragen, die für diesen Knoten bestimmt waren, anderen Knoten zuzuteilen.

- *Dienste an- oder abgemeldet:* Durch das Anmelden neuer Dienste eröffnen sich für $A_{Coordinator}$ auch neue Möglichkeiten, Anfragen zuzuweisen.

Je nach Algorithmus kann $A_{Coordinator}$ auch durch die Abmeldung eines Dienstes zu der Entscheidung kommen, eine Anfrage nicht länger zu puffern, sondern einem anderen Dienst zuzustellen.

Verlässt ein Knoten den Cluster, bevor er seine Dienste regulär abmelden konnte, so übernimmt dies LATE RISER, sobald es über den Vorgang informiert wurde.

- *Clusterknoten aufgewacht:* $A_{Coordinator}$ kann einem Clusterknoten sofort nach dessen Aufwachen Anfragen zusenden.

- *Anfrage angekommen:* Erreicht den Clusterhead eine neue Anfrage, so erfolgt eine Neuberechnung nach $A_{Coordinator}$ auch dann, wenn bereits eine Reihe Anfragen dieser Dienstklasse gepuffert werden. $A_{Coordinator}$ kann die Verteilung der Anfragen schließlich auch prioritätsbasiert oder nach der Länge von Warteschlangen vornehmen.

- *Anfrage beantwortet:* Wird auf eine Anfrage eine Antwort geben, so weiß der Clusterhead nach deren Erhalt, dass die Anfrage vollständig bearbeitet wurde. Dieses Wissen kann $A_{Coordinator}$ helfen, weitere Anfragen sinnvoll zu verteilen.

Beim Aufruf von $A_{Coordinator}$ informiert der Clusterhead den Algorithmus über das vorliegende Ereignis nach dieser Liste. $A_{Coordinator}$ kann selbst entscheiden, ob eine Neuberechnung notwendig ist oder nicht.

Metriken für die Verteilung der Anfragen $A_{Coordinator}$ braucht die Verteilung der Anfragen nicht nur auf der Grundlage der eben genannten Ereignisse zu entscheiden. Weiter stehen dem Algorithmus eine Reihe von Informationen über die Dienste und Knoten zur Verfügung, die der Clusterhead sammelt: Zwischenankunftszeiten von Anfragen, die Bearbeitungsdauer von Anfragen auf welche eine Antwort gegeben wird, Länge von Anfragen und Antworten und je Knoten und Dienstklasse die Anzahl der momentan bearbeiten Anfragen.

Falls eine Dienstklasse Sessions verwendet, muss $A_{Coordinator}$ dies natürlich berücksichtigen.

Für $A_{Coordinator}$ kommen Varianten gängiger Scheduling-Algorithmen in Frage, da sich die üblichen Anforderungen an Scheduler – minimale Antwort- und Wartezeiten, maximaler Durchsatz, Fairness [Pepper 1995, S. 332] – zum Großteil auch in den Anforderungen an LATE RISER, siehe Kapitel 3.3, finden lassen. Auch prioritätsbasierte Scheduler können bei gewissen Dienstklassen zum Einsatz kommen.

LATE RISER geht prinzipiell davon aus, dass ein Knoten beliebig viele Anfragen entgegennehmen kann. Diese müssen aber nicht alle parallel bearbeitet werden. $A_{Coordinator}$ sollte natürlich so gestaltet sein, dass ein Knoten nicht übermäßig belastet wird.

Metriken für die Wahl der Schlafphasen Wird keine dienstspezifische Implementierung von $A_{Coordinator}$ verwendet, so kann sich der Algorithmus bei der Wahl der Schlafphasen, wie bei der Verteilung der Anfragen, an verschiedenen Metriken und Informationen über die Dienste orientieren. Hierzu exemplarisch drei mögliche Ansätze:

1. *Die Schlafzeiten werden proportional zur durchschnittlichen Zwischenankunftszeit der Anfragen gewählt:* Wacht ein Knoten auf, so sind in der Regel ein, zwei Anfragen vorhanden, die er beantworten kann. Kein Knoten wird plötzlich mit besonders vielen Anfragen konfrontiert.

   Dieses Vorgehen hat aber auch eine Reihe von Nachteilen. Bei großen Zwischenankunftszeiten müssen Klienten oft lange warten. Bei sehr kurzen Zwischenankunftszeiten wird ein Dienstknoten hingegen kaum schlafen können, auch wenn die Bearbeitungszeiten fast Null sind. Bieten mehrere Knoten den gleichen Dienst, so erhält nicht abwechselnd ein Knoten jeweils alle Anfragen, sondern jeder Knoten ständig ein paar wenige.

2. *Die Schlafzeiten werden antiproportional zur Auslastung gewählt:* Nimmt die Zahl der Anfragen insgesamt zu, so werden die Knoten stärker belastet und sollten kürzer schlafen, um der Menge der Anfragen gewachsen zu sein.

   Dieses Vorgehen setzt allerdings voraus, dass die verschiedenen Knoten die gleichen Dienste anbieten. Andernfalls darf von der Gesamtauslastung des Clusters nicht auf die Auslastung eines einzelnen Knotens geschlossen werden.

   Vereinzelt kann es auch sinnvoll sein, einem bereits stark belasteten Knoten nur noch bestimmte Anfragen zuzuteilen und die Schlafphasen, entgegen der Vorgabe durch die Auslastung, *nicht* weiter zu verkürzen.

3. *Die Schlafzeit wird proportional zur Bearbeitungszeit einer Anfrage gewählt:* Bei Anfragen mit langer Bearbeitungszeit fallen dem Klienten Verzögerungen durch lange Schlafphasen nicht so unangenehm auf wie bei Anfragen mit sehr kurzer Bearbeitungszeit.

   Die Schlafzeit eines Knotens muss sich bei diesem Vorgehen an dem Dienst orientieren, der unter allen Dienstinstanzen des Knotens die kürzeste Bearbeitungszeit hat.

   Dieses Vorgehen hat zwei Nachteile. Bietet ein Knoten einen Dienst mit sehr kurzer Bearbeitungszeit, so wird er kaum schlafen können. Bieten mehrere Knoten den gleichen Dienst, so erhalten diese nicht abwechselnd alle anstehenden Anfragen, sondern ständig einige wenige.

Die beispielhaft diskutierten Ansätze zeigen, dass es nicht einfach ist, einen geeigneten Algorithmus $A_{Coordinator}$ zu entwerfen. Es können nicht nur weitere Diensteigenschaften sondern auch Knoteneigenschaften berücksichtigt werden. Um den Umfang dieser Arbeit nicht zu sprengen, wird diese Diskussion nicht weiter vertieft. Nur drei grundlegende Anforderungen an die Schlafphasen seien noch genannt:

1. *Maximalwert für die Schlafdauer:* $A_{Coordinator}$ sollte die mögliche Schlafdauer begrenzen. Dies garantiert, dass ein Knoten nach einer gewissen Zeit in jedem Fall wieder Anfragen entgegennehmen kann.

2. *Minimalwert für die Schlafdauer:* Zu kurze Schlafphasen sollten vermieden werden. Der Wechsel von einer Wach- in eine Schlafphase und umgekehrt ist in jedem Fall mit Transitionskosten verbunden – auch für den Clusterhead.

3. *Abstimmung von Schlafphasen:* Bieten mehrere Knoten den gleichen Dienst, so sollten ihre Schlafphasen aufeinander abgestimmt werden. Nur so sind längere Schlafphasen möglich, ohne die Wartezeit für die Klienten zu erhöhen.

Folgeanfragen Eine besondere Herausforderung sind die bereits bei SANDMAN diskutierten Folgeanfragen – Anfragen, die derselbe Klient auf eine Antwort hin sendet. LATE RISER betrachtet nur Folgeanfragen, die an dieselbe Dienstklasse gerichtet sind. Unabhängig davon ob Sessions verwendet werden oder nicht – Folgeanfragen sollten, wie in Kapitel 3.2 diskutiert, ohne Verzögerung durch Schlafphasen beantwortet werden.

Wann eine Anfrage eine Folgeanfrage ist, lässt sich ohne Kenntnis der klientseitigen Anwendung nicht sicher entscheiden. Folgeanfragen können daher nur angekündigt oder vermutet werden.

- *Ankündigen:* Beim Senden einer ersten Anfrage kündigt der Klient eventuelle Folgeanfragen explizit an. Beim Beispiel des Dateidienstes würde er dies beim Öffnen der Datei ankündigen. Treten Folgeanfragen – wie bei dem Dateidienst – sehr häufig auf, so kann dies auch vorab in der Dienstbeschreibung mitgeteilt werden, ähnlich dem in Kapitel 5.6 diskutierten Vorschlag aus [KraKri 2000, Kap. 5], dass Anwendungen Einfluss auf die Strategie nehmen können, nach der Schlafphasen gewählt werden.

  Nach Empfang einer Antwort und deren Weiterleitung an den Klienten muss ein passender Dienstknoten für eine gewisse Zeit T^*_{Fa} wach gehalten werden. Ist der Dienst zustandsbehaftet, kommt dafür nur der Dienstknoten in Frage, der auch die vorherige Anfrage beantwortet hat. T^*_{Fa} wird so gewählt, dass Folgeanfragen den Clusterhead innerhalb dieser Zeit erreichen können.

  Wird in jedem Fall eine Folgeanfrage gesendet, so kann $T^*_{Fa} = \infty$ gewählt werden. Der Dienstknoten kann erst dann in die Schlafphase wechseln, wenn die Folgeanfrage von ihm beantwortet oder einem anderen Knoten zugeteilt wurde.

- *Vermuten:* Der Clusterhead ermittelt selbstständig, wie häufig und nach welcher Zeit Anfragen desselben Klienten an denselben Stellvertreterdienst gesendet werden.

  Aus der Häufigkeit und dem zeitlichen Abstand berechnet er ein geeignetes T^*_{Fa} und verfährt dann wie eben beschrieben.

LATE RISER verwendet eine Variante des ersten Ansatzes. Bei der Beschreibung eines Dienstes kann T^*_{Fa} explizit mitgeteilt werden. Falls nicht, so wird $T^*_{Fa} = 0$ angenommen.

Der zweite Ansatz ist für LATE RISER zu aufwändig: Der Clusterhead müsste über kurz zurückliegende Antworten Buch führen und daraus die Häufigkeit von Folgeanfragen ermitteln. Treten relativ viele Folgeanfragen auf, so müsste er aus Parametern wie Mittelwert und Standardabweichung der zeitlichen Verzögerung, mit der die Folgeanfragen eintreffen, ein geeignetes T^*_{Fa} bestimmen.

6.6 Wahl des Clusterheads

Wie oben erwähnt, kann ein beliebiger Algorithmus zur Bildung von Clustern und zum Clustermanagement verwendet werden. Die Wahl des Clusterheads treffen viele energieeffiziente Algorithmen an Hand einer Metrik aus Energievorrat und den möglichen Funkverbindungen, wie bei CEC [YaBi+ 2003] und SPAN [BeJa+ 2001] in Kapitel 5.4 diskutiert. Weitere Faktoren können auch Eigenschaften der Hardware wie Rechenleistung oder Speicherkapazität sein.

LATE RISER übernimmt dieses Vorgehen. Nicht berücksichtigt werden die Dienste, welche die verschiedenen Knoten anbieten.

Ein weiterer Faktor sind Netzwerkschnittstellen eines Knotens. In dieser Arbeit werden vor allem die häufig anzutreffenden Funknetzschnittstellen betrachtet. LATE RISER geht davon aus, dass nur ein Netz mit gleichartigen Netzwerkschnittstellen existiert. Die Verwendung mehrerer Netze ist prinzipiell möglich, würde aber den Rahmen dieser Arbeit sprengen, da für eine sinnvolle Verwendung noch viel Forschungsarbeit geleistet werden muss.

Wurde ein Knoten zum Clusterhead gewählt, so übernimmt er diese Aufgabe in der Regel von einem anderen Knoten. Ausnahmen sind die Neubildung eines Clusters oder der Ausfall eines Clusterheads. Bei einer planmäßigen Neuwahl sorgt LATE RISER dafür, dass der alte Clusterhead dem neuen Clusterhead Verwaltungsinformationen wie Stellvertreterdienste und bestehende Sessions überträgt. Anstehende und laufende Anfragen zustandsloser Dienste können ebenfalls übertragen werden oder, falls der alte Clusterhead noch einige Zeit verfügbar ist, von diesem regulär bearbeitet und abgeschlossen werden.

Während die Clusterknoten vom Clustermanagement über den neuen Clusterhead informiert werden, ist dies für alle anderen Knoten Aufgabe des in Kapitel 6.2 diskutierten Discovery-Algorithmus'. So wird garantiert, dass alle Dienstinstanzen im Cluster und alle Klienten in- und außerhalb des Clusters über den Wechsel informiert werden und sich entsprechend darauf einstellen können. Sessions und laufende Anfragen müssen nicht abgebrochen werden.

7 Implementierung von LATE RISER

LATE RISER wurde in die bestehende Systemsoftware BASE integriert.[12] BASE ist eine kleine, flexible Middleware für Ubiquitous Computing und wurde im Rahmen des Projekts Peer to Peer Pervasive Computing [3PC] an der Universität Stuttgart entwickelt. BASE setzt auf der Java 2 Microedition-Plattform [J2ME] auf, die an verschiedenste Geräte angepasst werden kann.

Für die Implementierung und Simulation von LATE RISER wurde WebSphere Studio Device Developer [WSDD] von IBM verwendet. Eine Einführung in die Java-Programmierung mit WebSphere Studio Device Developer findet sich in [Gilhooly 2004].

[12] Entsprechend der Ausschreibung war zunächst eine Integration in die bestehende SANDMAN-Implementierung im Netzwerksimulator NS2 [NS2] geplant. Auf Vorschlag des Betreuers dieser Studienarbeit wurde stattdessen eine Integration in BASE vorgenommen. Die Implementierung in BASE ist die „vollständigere" Lösung – macht aber eine Messung des Energieverbrauchs von LATE RISER im Rahmen dieser Arbeit zu aufwändig. Dafür zeigt sie die Machbarkeit einer kommerziell nutzbaren Implementierung.

In der Simulation wird jeder Knoten durch eine eigene Instanz von BASE repräsentiert, die in einer separaten Java-Virtual-Machine ausgeführt wird. Die Knoten sind dadurch logisch so weit getrennt, dass sie nicht direkt mit Java-Aufrufen, sondern nur indirekt über die lokale Netzwerkschnittstelle kommunizieren können.

7.1 Eigenschaften von BASE

Ein wichtiges Merkmal von BASE ist die flexible Erweiterbarkeit. Dies wird besonders bei der Kommunikation zwischen Knoten deutlich.

Kommunikation Zur Kommunikation wird die Implementierung eines Schichtenmodells, ähnlich dem OSI-Referenzmodell, verwendet. Eine Darstellung der Schichten findet sich in Tabelle 3.

Semantik	(Semantic)
Interoperabilität	(Interoperability)
Transport	(Transport)
Sender-Empfänger	(Transceiver)

Tabelle 3: Schichtenmodell zur Kommunikation in BASE

Jede Schicht wird durch ein Plug-In repräsentiert und implementiert. Stehen für eine Schicht verschiedene Plug-Ins zur Verfügung, so können diese zur Laufzeit beliebig ausgetauscht werden. BASE ermittelt selbstständig an Hand von Knotenbeschreibungen und Anforderungen von Diensten, welche Plug-Ins für den Austausch einer *Invocation* – dem einzigen Nachrichtentyp in BASE – geeignet sind.

Die Plug-Ins werden zentral von einem Plug-In-Manager verwaltet. Für einen Sendevorgang durchläuft der Plug-In-Manager die Schichten des Kommunikationsmodells von oben nach unten und ruft jeweils das passende Plug-In auf. Beim Empfang einer Invocation werden die Plug-Ins entsprechend in umgekehrter Reihenfolge aufgerufen.

Proxy und Skeleton Für entfernte Aufrufe verwendet BASE die Konzepte gängiger Middleware. Sollen Schnittstellen einer Klasse entfernt verfügbar sein, so werden sie zusätzlich in einem Java-Interface beschrieben. Eine entsprechende Proxy-Klasse, die dieses Interface implementiert, setzt die lokalen Methodenaufrufe in entfernte Aufrufe um.

Dazu werden die Signatur der aufgerufenen Methode, die Parameter und einige andere Informationen in einer Invocation zusammengefasst. Der Proxy übergibt die Invocation dem Invocation-Broker, welcher sie an den Plug-In-Manager leitet. Dieser sorgt dafür, dass die Invocation durch das Interoperability-Plug-In serialisiert und vom Transceiver-Plug-In übertragen wird.

Nach dem Empfang und der Deserialisierung im Zielsystem wird es dem dortigen Invocation-Broker übergeben. Dieser ruft die entsprechende Skeleton-Klasse auf, welche die Invocation wieder in einen lokalen Aufruf umsetzt.

Für das Übertragen des Ergebnisses eines entfernten Aufrufs verwendet BASE die gleiche Prozedur.

Clustermanagement Ein Clustermanagement für BASE befand sich während der Entwicklung von LATE RISER noch in der Implementierungsphase. Daher wurde im Rahmen dieser Arbeit auf eine eigene umfassende Implementierung verzichtet und stattdessen ein rudimentärer Ansatz verwendet, der über Simulation getrieben wird.

Erkennen anderer Knoten Das Erkennen anderer Knoten ist in BASE Aufgabe des Discovery-Plug-Ins. Dieses existiert neben den oben genannten Plug-Ins. Durch das Discovery-Plug-In erfährt ein Knoten auch, welche Plug-Ins ein anderer Knoten unterstützt.

Die Knoten unterscheiden sich paarweise durch eine System-ID. Sie bildet eine eineindeutige Zuordnung zwischen den Knoten und den ganzen Zahlen. Die System-ID wurde in BASE bisher für jeden Knoten zufällig gewählt, bevor sie in Zukunft vom stabilen Speicher gelesen werden soll. Für LATE RISER wurde eine aufsteigende Zahlenfolge 1, 2, ... gewählt, damit die System-IDs für die Simulation des Clustermanagements vorab bekannt sind.

Diensterkennung Zur Diensterkennung verwendet BASE sequentielles Nachfragen bei allen erreichbaren Knoten. Entsprechende Mittel zur Beschreibung von Diensten und besonderen Dienstattributen waren bei der Implementierung von LATE RISER bereits vorhanden. In BASE können Dienste nicht nur entfernt genutzt, sondern auch instanziiert werden.

Eine Integration der Diensterkennung von LATE RISER, in die vorhandene, komplexe Diensterkennung von BASE, ist sehr aufwändig. Daher wurde für LATE RISER eine völlig neue Diensterkennung implementiert.

Verwendet wurde allerdings das Konzept der Objekt-IDs und der Objekt-Registrierung. Soll ein Java-Objekt entfernt aufgerufen werden können, so wird sein Skeleton bei der Objekt-Registrierung angemeldet. Diese ordnet dem Objekt und seinem Skeleton eine lokal eineindeutige Objekt-ID zu. An Hand der Objekt-ID kann der Invocation-Broker eine ankommende Invocation dem passenden Skeleton zuordnen.

System- und Objekt-ID bilden zusammen die Referenz-ID, welche ein Objekt im gesamten Netz eineindeutig bezeichnet.

Die Diensterkennung ist in BASE als so genannter Well-known-Service implementiert. Sie hat auf jedem Knoten die gleiche Objekt-ID und ist daher vorab bekannt. Für die Diensterkennung in LATE RISER wurde dieses Vorgehen übernommen.

Sessions Zum Zeitpunkt der Implementierung von LATE RISER unterstützte BASE drei Arten von Invocations:

- *Invoke:* Anfrage an einen Dienst.
- *Result:* Antwort eines Dienstes.
- *Remove:* Empfangsbestätigung für eine Antwort.

Auf eine Anfrage gibt BASE immer eine – gegebenenfalls leere – Antwort. Einweg-Nachrichten werden nicht explizit unterstützt.

Der Einfachheit halber wurde bei der Implementierung von LATE RISER gänzlich auf das in Kapitel 6.4 beschriebene Sessionkonzept verzichtet. Dienste werden in diesem Sinn als zustandslos angenommen. Auch Folgeanfragen werden in dieser Implementierung nicht gesondert behandelt.

7.2 Einbettung von LATE RISER

Zur Einbettung in BASE wurde LATE RISER in drei Pakete aufgeteilt. Diese sind in Abbildung 10 dargestellt und werden im Folgenden kurz erläutert.

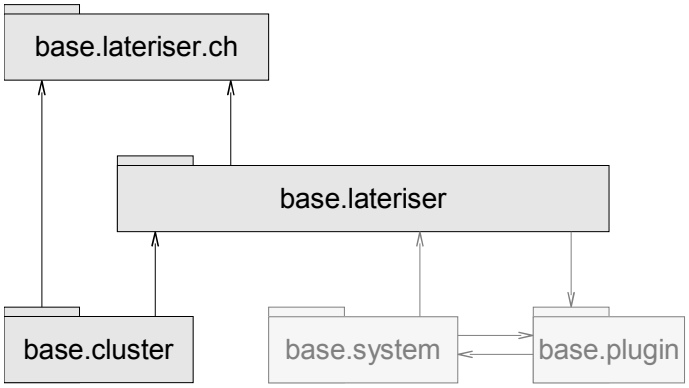

Abbildung 10: Einbettung von LATE RISER in BASE

- `base.lateriser`: Dieses Paket bietet Basisfunktionen von LATE RISER, die von jedem Knoten genutzt werden. Die zentrale Klasse `LateRiser` kapselt alle Methoden, die ein Knoten lokal benötigt.
- `base.lateriser.ch`: Dieses Paket bietet alle Algorithmen und Funktionen, die ein Clusterhead benötigt, um seinen Cluster zu koordinieren. Die zentrale Klasse `Coordinator` kapselt alle Methoden, die der Clusterhead lokal aufruft.
- `base.cluster`: Dieses Paket definiert die Schnittstellen zum Clustermanagement. Es bietet zwei Listener, mit denen LATE RISER über Änderungen in der Clusterkonfiguration informiert wird. Die zentrale Klasse `ClusterManagement` bietet in dieser Implementierung nur einige Getter und Setter für den Clusterhead und die Menge der Clusterknoten. Sie wird über Simulation getrieben.

7.3 Coordinator-Algorithmus

$A_{Coordinator}$ wurde in dieser Implementierung sehr einfach gewählt und es wurde keine dienstspezifische Anpassung vorgenommen. Beim Entwurf wurde vor allem auf folgende Anforderungen aus Kapitel 3.3 geachtet:

- Kurze Wartezeiten mit angeschalteter Netzwerkschnittstelle
- Rasche Erreichbarkeit

Durchschnittliche Bearbeitungszeit einer Anfrage $A_{Coordinator}$ kann, wie oben beschrieben, die Verteilung der Anfragen und die Wahl der Schlafphasen an Hand verschiedener Metriken und Informationen, die LATE RISER führt, vornehmen.

In dieser Implementierung verwendet $A_{Coordinator}$ die durchschnittliche Bearbeitungszeit $\overline{T}_B$ von Anfragen einer Dienstklasse. Die Bildung von Durchschnitten setzt voraus, dass gleiche Dienstinstanzen nur von Rechnern angeboten werden, die sich in Hard- und Software ähneln.

Die Bearbeitungszeit einer Anfrage wird aus Sicht des Clusterheads ermittelt. Dazu misst er die Zeit vom Senden einer Anfrage bis zum Eintreffen der Antwort. Da bei BASE auf jede Anfrage eine Antwort gegeben wird, müssen Dienste, die keine Antwort geben, nicht berücksichtigt werden.

Im zu Grunde liegenden Modell wird davon ausgegangen, dass die Bearbeitungszeit parallel gestellter Anfragen linear mit der Anzahl der Anfragen skaliert. In der Realität kann die Bearbeitungszeit natürlich auch sublinear mit der Anzahl der Anfragen steigen, zum Beispiel bei Verzögerungen durch Festplattenzugriffe. Ebenso ist aber auch ein superlinearer Anstieg möglich, wenn beispielsweise Swapping aktiviert wird. Bearbeitet ein Knoten alle Anfragen sequentiell, so passt das Modell der linearen Skalierung exakt.

Entsprechend berechnet der Clusterhead die Bearbeitungszeit T_B *einer* Anfrage, indem er die Zeit vom Senden der Anfrage bis zum Erhalt der Antwort zu jedem Zeitpunkt durch die Anzahl der momentan zugeteilten Anfragen dividiert. Ein einfaches Rechenbeispiel ist in Abbildung 11 dargestellt.

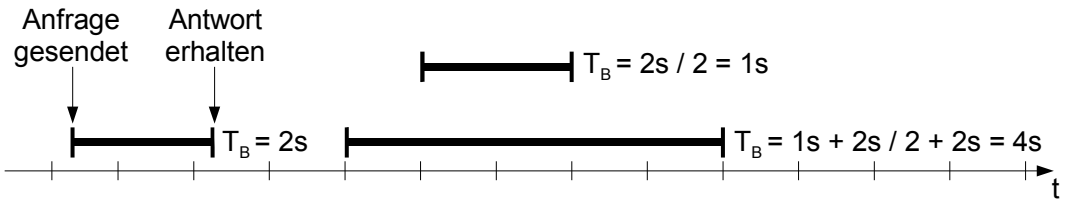

Abbildung 11: Berechnung der Bearbeitungszeiten dreier Anfragen

Aus den einzelnen Bearbeitungszeiten T_B berechnet jeder Stellvertreterdienst die durchschnittliche Bearbeitungszeit $\overline{T}_B$ für Anfragen seiner Dienstklasse. Da $\overline{T}_B$ lediglich als Richtwert und Entscheidungshilfe für $A_{Coordinator}$ dient, reicht dieses Vorgehen aus und steht in einem sinnvollen Verhältnis zwischen Nutzen und Aufwand.

Der Algorithmus im Detail Der hier gewählte Algorithmus $A_{Coordinator}$ gliedert sich in drei Abschnitte.

Im ersten Abschnitt, dargestellt in Abbildung 12, berechnet $A_{Coordinator}$ für jeden Knoten die Zeit, die dieser vermutlich benötigt, bis er eine Anfrage bearbeiten kann. Bei einem Knoten,

der schläft, ist dies die Zeit bis zu seinem Aufwachen. Bei einem Knoten, der wach ist, ist dies die Summe der durchschnittlichen Bearbeitungszeiten $\overline{T}_B$ aller Anfragen, die er gerade bearbeitet.

Anschließend werden die Knoten entsprechend der eben berechneten Zeiten sortiert. Auf diese Weise ermittelt $A_{Coordinator}$ jene Knoten, die Anfragen bei sequentieller Bearbeitung, entsprechend dem Modell der linear skalierenden Bearbeitungszeiten, am schnellsten beantworten werden.

```
ARRAY Knoten[] := Alle Knoten des Clusters inklusive Clusterhead;
ARRAY Stellvertreterdienste[] := Alle Stellvertreterdienste;
ARRAY belegt_Zeiten[] := NEW ARRAY über Knoten[];

FORALL k IN Knoten[] DO
  IF k ist wach THEN
    belegt_Zeiten[k] := 0;
    FORALL s IN Stellvertreterdienste[] DO
      n := Anzahl Anfragen von s, die k gerade bearbeitet;
      belegt_Zeiten[k] := belegt_Zeiten[k] + n * s.TB;
    END FOR
  ELSE
    belegt_Zeiten[k] := Zeit bis k aufwacht;
  END IF
END FOR

SORT(Knoten[] und belegt_Zeiten[] nach belegt_Zeiten[]);
```

Abbildung 12: Erster Abschnitt von $A_{Coordinator}$

Im zweiten Abschnitt arbeitet sich $A_{Coordinator}$ schrittweise durch die Anfrage-Warteschlangen bei den Stellvertreterdiensten, wie in Abbildung 13 dargestellt. Die älteste Anfrage wird zuerst betrachtet.

Für jede Anfrage durchläuft der Algorithmus die eben sortierte Knotenliste und ermittelt den ersten Knoten, der den passenden Dienst bietet. Falls dieser Knoten wach ist, wird ihm die Anfrage zugestellt. Schläft der Knoten im Moment, so wird die Anfrage in der Warteschlange belassen und der entsprechende Stellvertreterdienst als abgehandelt markiert.

Liegen bei einem Stellvertreterdienst keine Anfragen vor, so wird er bereits zu Beginn als abgehandelt markiert.

$A_{Coordinator}$ verteilt die Anfragen so lange, bis alle Stellvertreterdienste abgehandelt wurden.

Im dritten und letzten Abschnitt legt $A_{Coordinator}$ alle Clusterknoten schlafen, die momentan keine Anfrage bearbeiten. Je Clusterknoten ermittelt er den Dienst, der die kürzeste durchschnittliche Bearbeitungszeit $\overline{T}_B$ hat und verwendet als Schlafdauer $2\,\overline{T}_B$. Bietet ein Knoten überhaupt keinen Dienst, so wird er für eine fest vorgegebene Zeit schlafen gelegt, wie in Abbildung 14 gezeigt.

Offensichtlich werden die oben genannten Anforderungen erfüllt:

- *Kurze Wartezeiten mit angeschalteter Netzwerkschnittstelle:* $A_{Coordinator}$ teilt einem

```
ARRAY abgehandelt[] := NEW ARRAY über Stellvertreterdienste[];

FORALL s IN Stellvertreterdienste[] DO
  IF s.Warteschlage ist leer THEN
    abgehandelt[s] := TRUE;
  ELSE
    abgehandelt[s] := FALSE;
  END IF
END FOR

WHILE es existiert s mit abgehandelt[s] = FALSE DO
  s := Stellvertreterdienst mit ältester Anfrage in Warteschlange
       unter noch nicht abgehandelten Stellvertreterdiensten;
  a := älteste Anfrage aus s.Warteschlange;

  FORALL k IN Knoten[] FROM MIN TO MAX(belegt_Zeiten[]) DO
    IF k bietet Dienstinstanz für s THEN
      IF k ist wach THEN
        Sende a an k;
        Entferne a aus s.Warteschlange;
        belegt_Zeiten[k] := belegt_Zeiten[k] + s.TB;
        RE-SORT(Knoten[] und belegt_Zeiten[] nach belegt_Zeiten[]);
      ELSE
        abgehandelt[s] = TRUE;
      END IF
      BREAK FOR-LOOP;
    END IF
  END FOR
END WHILE
```

Abbildung 13: Zweiter Abschnitt von $A_{Coordinator}$

```
FORALL k IN Knoten[] DO
  IF k != Clusterhead AND k ist wach
                      AND k bearbeitet momentan keine Anfrage THEN

    VAR sleep_duration = MAXIMALE_SCHLAFZEIT;

    FORALL s in Stellvertreterdienste[] DO
      IF k bietet Dienstinstanz für s THEN
        sleep_duration = MIN(sleep_duration, 2 * s.TB);
      END IF
    END FOR
    FALL_ASLEEP(k, sleep_duration);
  END IF
END FOR
```

Abbildung 14: Dritter Abschnitt von $A_{Coordinator}$

Clusterknoten sofort nach dem Aufwachen Anfragen zu oder versetzt ihn wieder in die Schlafphase.

Clusterknoten, die in der Rolle eines Klienten eine Antwort empfangen haben, werden von $A_{Coordinator}$ sofort in die Schlafphase versetzt, wenn sie nicht selber in der Rolle eines Dienstknotens auch Anfragen bearbeiten.

- *Rasche Erreichbarkeit:* $A_{Coordinator}$ versucht sofort nach dem Eintreffen einer Anfrage, diese einer entsprechenden Dienstinstanz zuzuteilen. Auch Antworten werden sofort an den eigentlichen Klienten weitergeleitet.

Unter den momentan wachen Knoten werden die Anfragen gleichmäßig nach Bearbeitungszeiten verteilt und dadurch keiner übermäßig belastet.

7.4 Exemplarische Dienstregistrierung, -erkennung und -nutzung

Abbildung 15 stellt den Ablauf einer Dienstregistrierung, -erkennung und -nutzung exemplarisch dar und zeigt in UML-Notation die benötigten Klassen und Pakete der LATE RISER-Implementierung. Drei Knoten sind an der Dienstnutzung beteiligt:

- *Knoten K* führt eine Anwendung aus, die einen entfernten Dienst aufruft.
- *Knoten D* bietet den entsprechenden Dienst. Da der Knoten an LATE RISER teilnimmt, kann er nur über seinen Clusterhead angesprochen werden.
- *Knoten C* ist Clusterhead des Clusters, in dem sich Knoten D befindet. Knoten K muss nicht zu diesem Cluster gehören.

In zwölf Schritten werden nun die verschiedenen lokalen und entfernten Aufrufe erläutert. In Abbildung 15 sind sie als Pfeile dargestellt und entsprechend nummeriert. Die ersten fünf Schritte gehören zur Dienstregistrierung, der nächste Schritt gehört zur Diensterkennung und die verbleibenden sechs Schritte gehören zur Dienstnutzung.

Dienstregistrierung

1. *Lokales Registrieren der Dienstinstanz:* Knoten D registriert seinen Dienst lokal unter dem Namen `Beispieldienst` bei LATE RISER. Dazu ruft die Dienstinstanz eine passende Methode der Klasse `LateRiser` auf. Diese legt einen entsprechenden Eintrag in der Instanz-Registrierung, ein Objekt der Klasse `InstanceRegistry`, an.

2. *Entferntes Registrieren der Dienstinstanz:* Die Instanz-Registrierung sorgt automatisch dafür, dass die Dienstinstanz auch beim Clusterhead unter dem Namen `Beispieldienst` registriert wird.

   Die Koordinierung des Clusters im Sinne von LATE RISER, übernimmt seitens des Clusterheads eine Instanz der Klasse `Coordinator`. `Coordinator` bietet eine entfernt aufrufbare Methode `addServiceInstance`, mit der die Registrierung der Dienstinstanz vorgenommen werden kann.

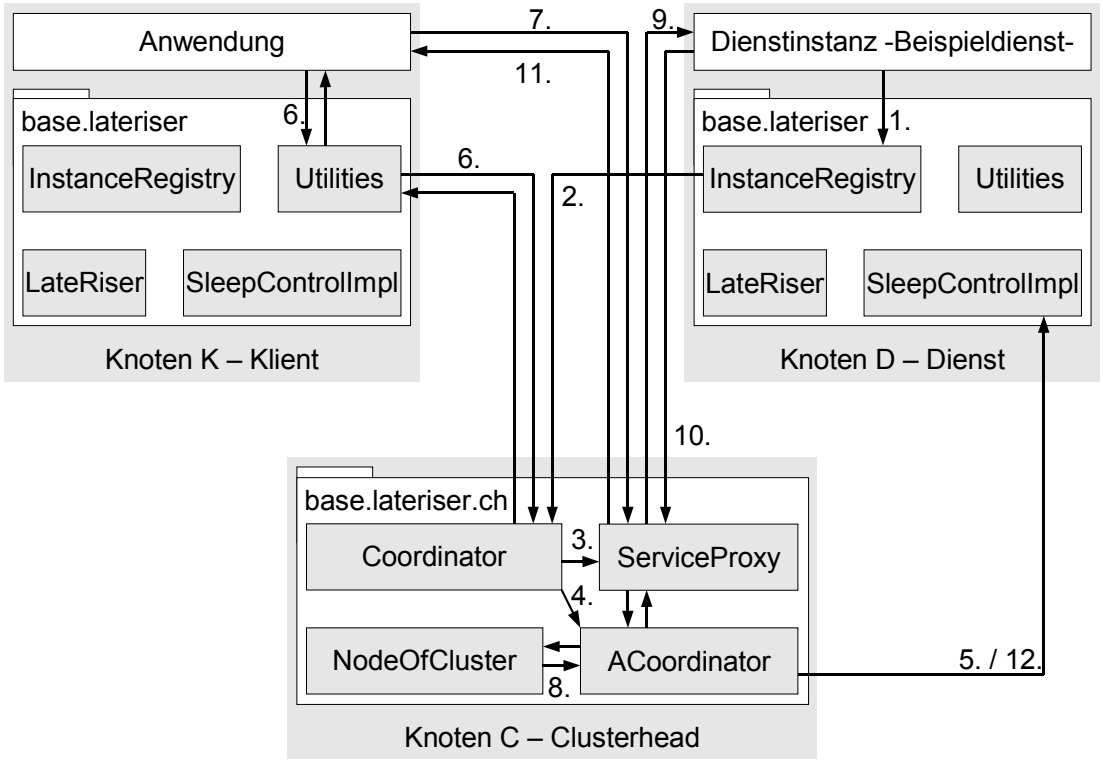

Abbildung 15: Dienstregistrierung, -erkennung und -nutzung in der Implementierung

3. *Stellvertreterdienst:* Das `Coordinator`-Objekt legt für die Dienstklasse `Beispieldienst` ein eigenes `ServiceProxy`-Objekt an. Die Dienstinstanz von Knoten D wird bei diesem registriert.

   Ein `ServiceProxy`-Objekt ist die Implementierung eines Stellvertreterdienstes.

4. *$A_{Coordinator}$ aufrufen:* Das `Coordinator`-Objekt ruft anschließend $A_{Coordinator}$ auf. Dieser Algorithmus wird durch eine Instanz von `ACoordinator` implementiert.

5. *Schlafenlegen:* Da momentan keine Anfragen an `Beispieldienst` oder andere Dienste von Knoten D vorliegen, wird dieser für einige Sekunden schlafen gelegt. Der Vorgang wird im `NodeOfCluster`-Objekt, das den Knoten D beim Clusterhead repräsentiert, vermerkt.

Diensterkennung

6. *Referenz zum Stellvertreterdienst:* Knoten K möchte eine Anfrage an `Beispieldienst` stellen. Daher bittet er mit Methoden der Klasse `Utilities` den Clusterhead um eine Referenz auf `Beispieldienst`. Der Clusterhead antwortet mit einer Referenz auf das entsprechende `ServiceProxy`-Objekt – den Stellvertreterdienst.

Dienstnutzung

7. *Anfrage senden:* Die Anwendung von Knoten K formuliert daraufhin eine Anfrage in Form einer Invocation an `Beispieldienst`. Knoten K kann das zugehörige `Invocation`-Objekt dank der eben erhaltenen Referenz an den Stellvertreterdienst senden.

   Der Stellvertreterdienst legt die Anfrage in seiner Warteschlage ab und informiert das `Coordinator`-Objekt darüber. Dieses ruft den Algorithmus $A_{Coordinator}$ auf. Da Knoten D – der einzige Knoten, der Anfragen an `Beispieldienst` beantworten kann – im Moment schläft, passiert zunächst nichts.

8. *Knoten D wacht auf:* Nach Ablauf der Schlafphase von Knoten D informiert das `NodeOfCluster`-Objekt den Algorithmus $A_{Coordinator}$ darüber, dass Knoten D eben aufgewacht ist.

   $A_{Coordinator}$ teilt die Anfrage von Knoten K der Dienstinstanz auf Knoten D zu, indem er die Methode `assignRequest` des `ServiceProxy`-Objekts von `Beispieldienst` aufruft.

9. *Zustellen der Anfrage:* Die Anfrage wird Knoten D in einer Kopie des ursprünglichen `Invocation`-Objekts zugestellt und von der Dienstinstanz bearbeitet.

   Im `NodeOfCluster`-Objekt, welches Knoten D repräsentiert, wird vermerkt, dass der Knoten momentan eine Anfrage bearbeitet.

10. *Antwort erreicht den Clusterhead:* Das Eintreffen der Antwort von Knoten D wird ebenfalls dem entsprechenden `NodeOfCluster`-Objekt mitgeteilt, welches daraus die Bearbeitungszeit T_B der Anfrage berechnet. T_B wird für die Berechnung der durchschnittlichen Bearbeitungszeit $\overline{T}_B$ im `ServiceProxy`-Objekt zu `Beispieldienst` gespeichert.

11. *Antwort weiterleiten:* Außerdem wird die Antwort im ursprünglichen `Invocation`-Objekt vom Clusterhead an Knoten K gesendet. Aus Sicht des Klienten ist die Bearbeitung der Anfrage damit abgeschlossen.

12. $A_{Coordinator}$ *aufrufen:* Beim Clusterhead endet die Bearbeitung der Anfrage mit einem Aufruf von $A_{Coordinator}$. Da momentan keine weiteren Anfragen vorliegen, die Knoten D zugeteilt werden können, wird dieser wieder schlafen gelegt.

7.5 Clustermanagement und -simulation

Zum Zwecke der Referenz werden in den folgenden Kapiteln die drei Pakete, die bei der Implementierung von LATE RISER zu BASE hinzugefügt wurden, einzeln erläutert.

Erste Neuerung ist das Paket `base.cluster`. Es bildet die Schnittstelle zwischen LATE RISER und einem zukünftigen Clustermanagement. Die zentrale Klasse `ClusterManagement` wird vor dem Start von LATE RISER über eine statische Methode, entsprechend dem Entwurfsmuster Singleton, einmal instanziiert. Das `ClusterManagement`-Objekt übernimmt zwei Aufgaben:

1. *Speichern der Clusterkonfiguration:* Es speichert für den Knoten, auf dem es ausgeführt wird, die System-ID des Clusterheads. Wie in Kapitel 6.1 beschrieben, ist ein einzelner Knoten sein eigener Clusterhead.

   Für einen Knoten, der momentan Clusterhead ist, speichert es zusätzlich die System-IDs der Clusterknoten, die zum Cluster gehören.

2. *Informieren über Änderungen:* Mit zwei Event-Listenern informiert das `ClusterManagement`-Objekt LATE RISER über Änderungen an der Clusterkonfiguration. Der `ClusterheadChangedListener` teilt die System-ID eines neuen Clusterheads mit. Der `ClusternodeChangedListener` ist nur für Verwendung auf Clusterheads konzipiert und meldet Knoten, die dem Cluster beigetreten sind oder ihn verlassen haben.

Zusätzlich bietet `ClusterManagement` Methoden an, mit denen die System-ID des Clusterheads und die System-IDs der zugehörigen Clusterknoten gesetzt werden können. Die verschiedenen Test- und Beispielanwendungen simulieren das Clustermanagement, indem sie diese Methoden auf allen Knoten und mit denselben Parametern synchron aufrufen.

Da jeder Knoten durch eine separate Instanz der Java-Virtual-Machine repräsentiert wird, ist maschinenübergreifende Synchronisation notwendig. In dieser Implementierung schreibt der Knoten mit der System-ID 1 seine Startzeit in eine globale Datei. Knoten mit größerer System-ID lesen diese Zeit aus und passen sich daran an. Dieses Vorgehen reicht aus, da in dieser Implementierung alle Knoten vom selben Rechnersystem simuliert werden.

7.6 Basisfunktionen von LATE RISER

Die Basisfunktionen von LATE RISER – sie werden von jedem Knoten verwendet – sind im Paket `base.lateriser` implementiert. Die Klassen des Pakets sind in Abbildung 16 dargestellt und werden im Folgenden einzeln erläutert.

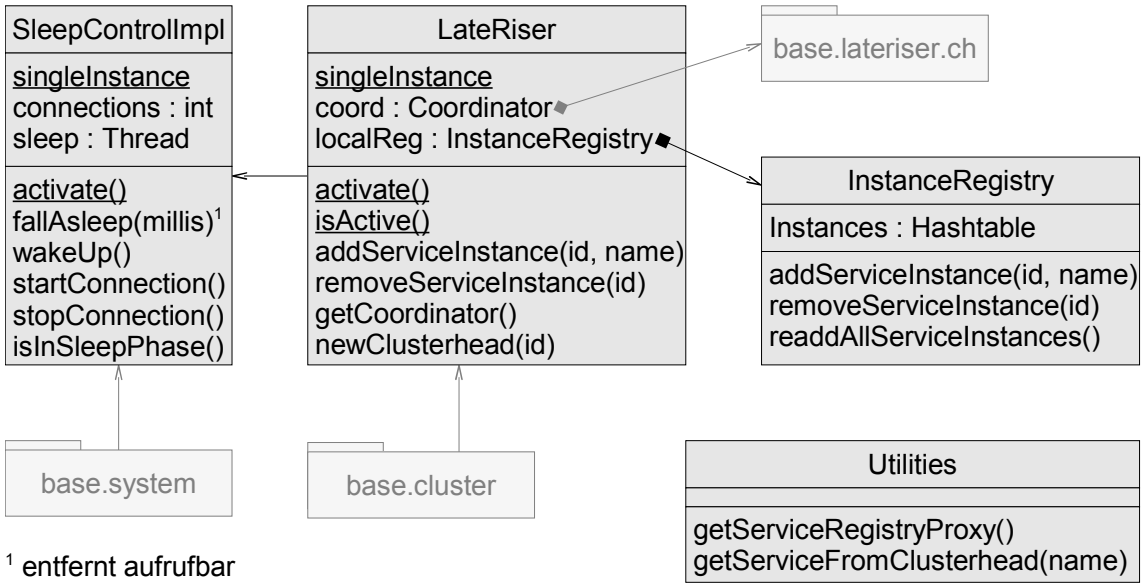

Abbildung 16: Klassendiagramm des Java-Pakets base.lateriser

base.lateriser.LateRiser Die Klasse `LateRiser` kapselt alle Funktionen zur Aktivierung von LATE RISER und zur Verwaltung von Diensten.

Nach Systemstart wird mit der statischen Methode `activate` eine Instanz von `LateRiser` angelegt, die dann über einen statischen Getter überall verfügbar ist. Mit `activate` wird auch eine Instanz der Klasse `InstanceRegistry` angelegt.

base.lateriser.InstanceRegistry Das `InstanceRegistry`-Objekt – die Instanz-Registrierung – verwaltet alle lokalen Dienstinstanzen, die mit LATE RISER entfernt aufgerufen werden können.

Lokale Dienstinstanzen können mit `addServiceInstance` unter einer lokal eineindeutigen Objekt-ID und einem frei wählbaren Namen angemeldet werden.

Das `InstanceRegistry`-Objekt meldet neue Dienstinstanzen unverzüglich dem Clusterhead. Wechselt dieser, so sorgt es mit dem Befehl `readdAllServiceInstances` dafür, dass der neue Clusterhead über alle Dienstinstanzen, die der Knoten anbietet, informiert wird.

base.lateriser.Utilities Die Klasse `Utilities` bietet statische Methoden, mit denen Anwendungen bei einem Clusterhead die Referenz-IDs entfernter Dienste erfragen können. Der Clusterhead antwortet mit der Referenz-ID des entsprechenden Stellvertreterdienstes.

Diese Methoden können auch von Knoten verwendet werden, die nicht an LATE RISER teilnehmen.

base.lateriser.SleepControlImpl Ein Clusterknoten kann von seinem Clusterhead über das Interface `SleepControl` in eine Schlafphase versetzt werden. Dazu ist entfernt die Methode `fallAsleep` verfügbar.

`SleepControlImpl`, die Implementierung von `SleepControl`, wird statisch mit der Klasse `LateRiser` aktiviert. Sendet ein lokales Objekt eine Invocation nach außen, so meldet dies der Invocation-Broker mit der Methode `startConnection` an `SleepControlImpl`. Wurde die Invocation gesendet, so ruft der Invocation-Broker `stopConnection` auf. Bei Invocations, auf die eine Antwort erwartet wird, wird erst nach deren Erhalt `stopConnection` aufgerufen.

`SleepControlImpl` ist auf diese Weise über alle ausgehenden Kommunikationsverbindungen informiert. Nur wenn keine ausgehende Kommunikationsverbindung existiert und der Knoten mit `fallAsleep` in die Schlafphase versetzt wurde, werden die Netzwerkschnittstellen tatsächlich deaktiviert.

Das Deaktivieren und Reaktivieren der Netzwerkschnittstellen wird durch zwei Methoden `fallAsleep` und `wakeUp` im Interface `Transceiver` ermöglicht.

In der Simulation werfen die Transceiver-Plug-Ins im deaktivierten Zustand beim Senden oder Empfangen einer Invocation eine entsprechende Fehlermeldung. Der Empfang einer Invocation weist nicht in jedem Fall auf einen Fehler in der Implementierung von LATE RISER hin. In der Realität müsste der Sender die Nachricht aber wiederholen.

Letzte Aufgabe der Instanz von `LateRiser` ist das Kreieren eines `Coordinator`-Objekts, sobald sie vom Clustermanagement über die Methode `newClusterhead` erfährt, dass der eigene Knoten Clusterhead geworden ist.

7.7 LATE RISER im Koordinator-Betrieb

Das Paket `base.lateriser.ch` übernimmt nach Instanziierung der zentralen Klassen `Coordinator` die Koordinierung des Clusters. Die Klassen des Pakets sind in Abbildung 17 dargestellt.

base.lateriser.ch.Coordinator Die Klasse `Coordinator` ist die zentrale Klasse des Pakets. Sie kapselt alle lokal und entfernt aufrufbaren Methoden, die für die Koordinierung des Clusters benötigt werden.

Entfernt und lokal stellt sie Methoden zum An- und Abmelden von Dienstinstanzen zur Verfügung. Diese Methoden werden von den Instanz-Registrierungen der anderen Knoten aufgerufen. Zur Diensterkennung bietet `Coordinator` eine Methode `getService`, die über die Klasse `Utilities` auch entfernt leicht aufgerufen werden kann.

`Coordinator` implementiert die Methoden des `ClusternodeChangedListener`-Interfaces, mit denen ihr das Clustermanagement Änderungen an der Clusterkonfiguration mitteilt.

base.lateriser.ch.NodeOfCluster Zu Verwaltungszwecken legt das `Coordinator`-Objekt für jeden Clusterknoten ein `NodeOfCluster`-Objekt an. Auch der Clusterhead repräsentiert sich selbst durch ein solches Objekt.

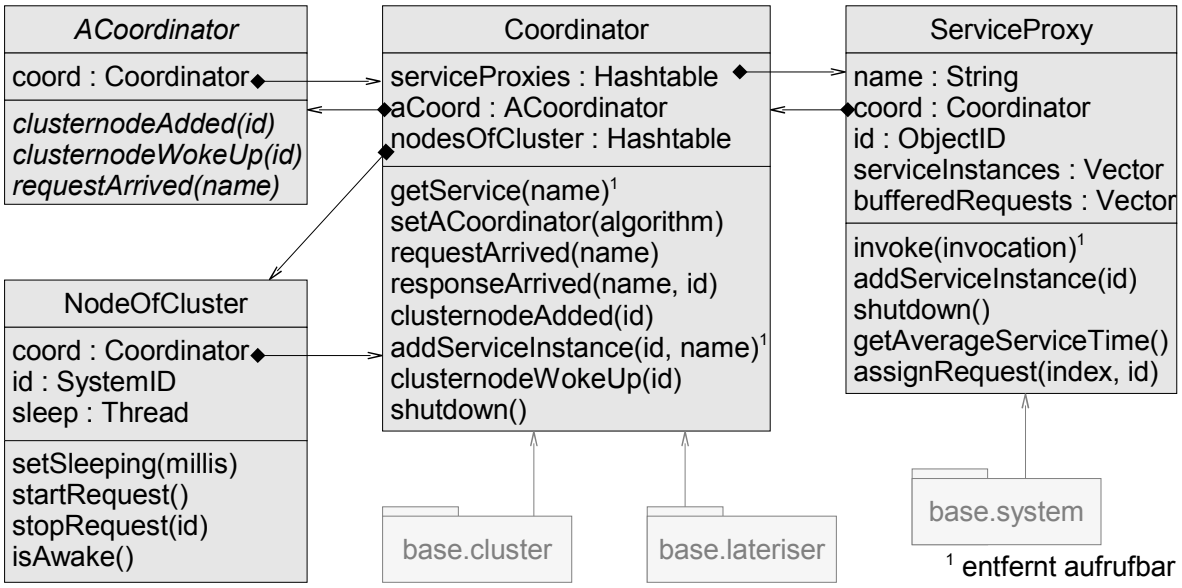

Abbildung 17: Klassendiagramm des Java-Pakets base.lateriser.ch

In den `NodeOfCluster`-Objekten wird die System-ID des entsprechenden Knotens, sein Zustand – schlafend oder wach – und die Anzahl der Anfragen, die er momentan bearbeitet, vermerkt. Ferner kann vermerkt und abgefragt werden, wann ein Knoten in die Schlafphase gesendet wurde und wie lange er noch schlafen wird.

Sobald ein `NodeOfCluster`-Objekt an Hand eines internen Zeitgebers registriert, dass der Knoten, den es repräsentiert, eben aufgewacht ist, informiert es das `Coordinator`-Objekt darüber.

base.lateriser.ch.ServiceProxy Eine Dienstklasse und die zugehörigen Dienstinstanzen werden durch ein `ServiceProxy`-Objekt repräsentiert. Es bildet den Stellvertreterdienst für eine Dienstklasse.

Dienstinstanzen der gleichen Dienstklasse werden an Hand des Namens erkannt, der bei der Anmeldung an der lokalen Instanz-Registrierung angegeben wird. Wird dem `Coordinator`-Objekt eine neue Dienstinstanz mitgeteilt, so ordnet es diese an Hand des Namens dem passenden `ServiceProxy`-Objekt zu. Bei Bedarf wird ein neues `ServiceProxy`-Objekt erstellt.

Um sich entfernten Systemen wie ein Dienst zu präsentieren, implementiert jedes `ServiceProxy`-Objekt das Interface `InvocationHandler`. Mittels dessen Methode `invoke` kann es Anfragen in Form eines `Invocation`-Objekts entgegennehmen und verarbeiten.

Erreicht ein `ServiceProxy`-Objekt eine neue Anfrage, so wird intern zunächst eine Beschreibung der Anfrage angelegt und der Thread, der die Anfrage mitgeteilt hat, angehalten. Das `ServiceProxy`-Objekt informiert das `Coordinator`-Objekt über die neue Anfrage. Dieses sorgt mit der unten beschriebenen Klasse `ACoordinator` dafür, dass die Anfrage einer passenden Dienstinstanz zugeteilt wird. Dem `ServiceProxy`-Objekt wird dies mit der Methode

`assignRequest` mitgeteilt. Anschließend wird der Thread, der die Anfrage mitgeteilt hat, fortgesetzt. Er übermittelt die Anfrage an die von `ACoordinator` gewählte Dienstinstanz, wartet auf die Antwort, nimmt diese entgegen und überträgt sie an den ursprünglichen Klienten.

Vor dem Senden der Anfrage an den eigentlichen Dienstknoten meldet das `ServiceProxy`-Objekt die Anfrage beim `NodeOfCluster`-Objekt an, das den Knoten repräsentiert, der die Anfrage bearbeiten wird. Nach Empfang der Antwort wird die Anfrage entsprechend wieder abgemeldet und das `NodeOfCluster`-Objekt teilt mit, wie lange die Bearbeitung der Anfrage gedauert hat. Berücksichtigt werden parallel gestellte Anfragen, wie Kapitel 7.3 erläutert. An Hand der verschiedenen Bearbeitungszeiten T_B berechnet jedes `ServiceProxy`-Objekt die durchschnittliche Bearbeitungszeit $\overline{T}_B$ für eine Anfrage seiner Dienstklasse.

base.lateriser.ch.ACoordinator Die Klasse `ACoordinator` ist eine abstrakte Klasse und repräsentiert alle Algorithmen $A_{Coordinator}$. Ein konkreter Algorithmus wird als Subklasse von `ACoordinator` implementiert und in Form eines Objekts an das `Coordinator`-Objekt übergeben.

`ACoordinator` bietet eine Reihe abstrakter Methoden, mit denen das `Coordinator`-Objekt sein `ACoordinator`-Objekt über Änderungen an der Clusterkonfiguration, das Aufwachen von Knoten, Änderungen an den Dienstinstanzen und das Eintreffen von Anfragen oder Antworten informieren kann.

Eine konkrete Implementierung von `ACoordinator` reagiert darauf, indem es Anfragen geeigneten Dienstinstanzen – damit auch Knoten – zuteilt und andere Knoten schlafenlegt.

Anfragen werden, wie eben beschrieben, mit `assignRequest` zugewiesen. Zum Schlafenlegen von Clusterknoten bietet `ACoordinator` eine Methode `sendClusternodeAsleep`, die einen Knoten über dessen `SleepControl`-Interface in die Schlafphase versetzt und diese im zugehörigen `NodeOfCluster`-Objekt lokal vermerkt.

Im Rahmen dieser Arbeit wurden drei konkrete Implementierungen von ACoordinator vorgenommen:

1. *ACoordinatorEmpty:* Diese Klasse repräsentiert eine leere Implementierung. Sie ignoriert alle Methodenaufrufe und existiert lediglich zu Testzwecken.

2. *ACoordinatorRandom:* Der Algorithmus dieser Klasse verteilt Anfragen zufällig an in Frage kommende Dienstinstanzen. Knoten werden nicht schlafen gelegt. Auch diese Klasse existiert vor allem zu Testzwecken.

3. *ACoordinatorSimple:* Diese Klasse ist eine Implementierung des in Kapitel 7.3 beschriebenen Algorithmus'. Da eine Berechnung nach diesem Algorithmus einige Zeit in Anspruch nimmt, wird er nicht mehr als viermal pro Sekunde ausgeführt. Dazu wird der eigentliche Algorithmus erst 50 bis 250 Millisekunden nach dem Aufruf in einem extra Thread gestartet. Durch die Verzögerung um mindestens 50 Millisekunden werden gehäuft auftretende Änderungen zusammen in einem Lauf von $A_{Coordinator}$ berücksichtigt.

7.8 Besonderheiten der Implementierung

In diesem Kapitel werden zwei besondere Aspekte der Implementierung von LATE RISER diskutiert.

Deserialisierung von Anfragen und Antworten Anfragen und Antworten werden in BASE durch `Invocation`-Objekte repräsentiert. Die Attribute eines solchen `Invocation`-Objekts enthalten unter anderem die System-IDs von Sender und Empfänger, die Signatur der aufgerufenen Methode, Parameter für den Aufruf, gegebenenfalls die Antwort und eventuell aufgetretene Ausnahmen, so genannte Exceptions.

Aufrufparameter, Antwort und Exceptions sind dabei dienstspezifisch und können Klassen verwenden, die nicht global bekannt sind. Klient- und Dienstknoten müssen diese Klassen natürlich kennen. Es darf allerdings nicht vorausgesetzt werden, dass ein Clusterhead diese Klassen kennt.

Der Clusterhead muss eine eintreffende Anfrage – zunächst ein Bytestrom – deserialisieren, das heißt, den Bytestrom in die ursprünglichen Java-Objekte zurückverwandeln, damit er die Anfrage für eine Weiterleitung an den eigentlichen Dienst entsprechend modifizieren kann. Gleiches gilt für Antworten. Zum Deserialisieren eines Objekts benötigt man die Klassen *aller* Attribute, die das Objekt besitzt; andernfalls ist eine Deserialisierung nicht möglich.

Da der Clusterhead die Klassen der dienstspezifischen Attribute eventuell *nicht* kennt, wurde bei der Implementierung von LATE RISER die Klasse Invocation modifiziert. Alle dienstspezifischen Attribute werden intern bereits serialisiert als Byte-Felder gespeichert. Nur bei Zugriff oder Änderung werden sie deserialisiert beziehungsweise serialisiert.

Zusammen mit den anderen Attributen eines Invocation-Objekts wird aus diesen Byte-Feldern dann bei Bedarf ein serialisiertes Invocation-Objekt erzeugt. Empfängt der Clusterhead nun eine Invocation, so deserialisiert er das Invocation-Objekt an sich, nicht aber in einem zweiten Schritt die dienstspezifischen Attribute. Auf diese Weise kann der Clusterhead die Invocation kopieren und modifizieren, ohne die dienstspezifischen Klassen zu kennen.

Konsistente Sichten Der Clusterhead benötigt während vieler Vorgänge eine konsistente Sicht auf seinen Cluster. Die ihm bekannte Clusterkonfiguration muss sich beispielsweise mit den Knoten decken, die bei ihm Dienstinstanzen registriert haben – auch wenn im Moment Dienstinstanzen an- oder abgemeldet werden. Während einer Berechnung nach $A_{Coordinator}$ dürfen sich weder die Clusterkonfiguration noch die registrierten Dienste ändern.

Ebenso benötigt die Instanz-Registrierung eines beliebigen Knotens während des An- oder Abmeldens einer Dienstinstanz eine konsistente Sicht des Clusterheads.

Um eine solche Sicht zu erhalten, sperrt diese Implementierung von LATE RISER die entsprechenden Objekte. Java bietet mit seinem Monitor-Konzept dafür geeignete Hilfsmittel.

Drei grobgranulare Sperren – auf dem `LateRiser`-, dem `Coordinator`- und dem `SleepControlImpl`-Objekt – werden verwendet. Die Aufteilung des Pakets `base.lateriser` in zwei unabhängig gesperrte Objekte `LateRiser` und `SleepControlImpl` ist unverzichtbar, will man hier verteilte Deadlocks verhindern.

Die Vielzahl paralleler Threads und die zyklischen Abhängigkeiten zwischen einigen Klassen des Pakets `base.lateriser.ch`, die leider unvermeidbar scheinen, verhindern ein übersichtliches hierarchisches Sperrkonzept. Die Verwendung grobgranularer Sperren bietet ein überschaubares Verfahren, das wegen nicht genutzter Parallelität aber Leistungseinbußen mit sich bringt.

8 Analyse von LATE RISER durch Simulation

Eine vollständige Messung der LATE RISER-Implementierung ist im Rahmen dieser Arbeit zu aufwändig. Stattdessen wurden in drei Simulationen die wichtigsten Eigenschaften von LATE RISER getestet und zentrale Messwerte bestimmt. Die Ergebnisse der Simulationen sind im Folgenden festgehalten. Da eine exakte Messung des Energieverbrauchs, ohne größere Änderungen an BASE, nicht möglich ist, werden hier nur die Dauer der Schlafphasen und Wachzeiten betrachtet.

In jeder Simulation wurde eine von zwei Testumgebungen verwendet, die im Zuge der Implementierung von LATE RISER entwickelt wurden. Jede Testumgebung stellt ein Szenario aus Clustern und Knoten dar und verwendet Dienste mit einem Java-Interface `ReverseString`. Die Implementierung von `ReverseString` dreht einen String um und liefert das Ergebnis zurück. Intern verwendet sie eine Sperre, so dass alle Anfragen sequentiell bearbeitet werden müssen. Durch eine `sleep`-Anweisung wird die Diensterbringung künstlich verzögert. Von außen scheint eine Dienstinstanz dadurch für mehrere Sekunden ausgelastet, auch wenn sie die CPU tatsächlich überhaupt nicht nutzt. Auf diese Weise können viele scheinbar voll ausgelastete Knoten auf einer einzelnen CPU simuliert werden.[13]

Jeder Knoten wurde in den folgenden Simulationen durch eine separate Java-Virtual-Machine dargestellt. Diese wurden allerdings alle auf ein und demselben Computer ausgeführt.

8.1 Schlafdauer und Wachzeit bei Leerlauf

Für die erste Simulation wurde eine Testumgebung mit sieben Knoten verwendet. Fünf Knoten, im Folgenden Server 1 bis 5 genannt, bieten je eine `ReverseString`-Implementierung als Dienst an, nutzen sie aber nicht. Die beiden anderen Knoten, fortan Klienten 1 und 2 genannt, bieten umgekehrt keinen Dienst an, nutzen aber den genannten Dienst.

Ziel der Simulation In dieser Simulation sollte die Wachzeit analysiert werden, die durch das regelmäßige Aufwachen und wieder Schlafenlegen entsteht. Ein wichtiger Parameter ist dabei die Dauer der einzelnen Schlafphasen.

Durchführung der Simulation In dieser Simulation wurden die Klienten passiv gewählt, das heißt, sie haben überhaupt keine Anfragen gestellt – das System ist so gesehen im Leerlauf.

Die Testumgebung wurde in dieser Konfiguration zehn Minuten simuliert. Zunächst bildete jeder Knoten einen eigenen Cluster. Nach 15 Sekunden wurden zwei Cluster gebildet,

[13] Bei den hier durchgeführten Simulationen bewegte sich die tatsächliche CPU-Auslastung, nach Angaben des Windows Task-Managers, in der Regel zwischen 0 und 20%.

bestehend aus den Servern 1 bis 4 beziehungsweise dem Server 5 und den Klienten 1 und 2. Im ersten Cluster war Server 4 der Clusterhead, im zweiten Server 5. Nach weiteren 30 Sekunden wurden die Knoten des ersten Clusters dem zweiten hinzugefügt.

Da die Klienten keine Dienste anboten, wurden sie jeweils für die maximal mögliche Schlafdauer – hier zehn Sekunden – schlafen gelegt. Da keine Dienstnutzungen stattfanden und der Clusterhead keine Werte T_B ermitteln konnte, verwendete er für die Berechnung der Schlafdauer der Server stets den vorgegebenen Wert $\overline{T}_B = 1$ Sekunde. Entsprechend wurden die Server für jeweils $2\,\overline{T}_B = 2$ Sekunden schlafen gelegt.

Die Simulation wurde dreimal durchgeführt. Anschließend wurde überprüft, dass die Messwerte stabil sind, das heißt im Vergleich zu den vorherigen Durchführungen sehr ähnlich oder gleich. Statt Mittelwerten wurden für die weitere Analyse nur die Messwerte der dritten und letzten Durchführung verwendet, um zu zeigen, dass auch diese schon sehr glatt wirken.

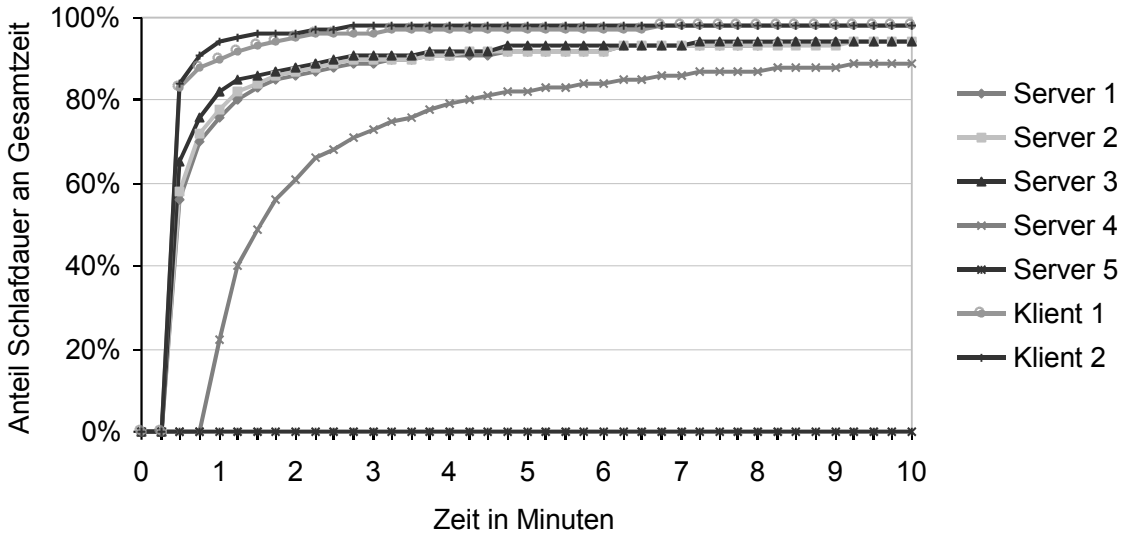

Abbildung 18: Relative Schlafdauer bei sieben Knoten *ohne* Dienstnutzung

Ergebnisse Abbildung 18 stellt den Zeitanteil, in dem ein Knoten schlief, über der Zeit dar. Leicht lässt sich erkennen, dass Server 5 nie geschlafen hat, da er ständig Clusterhead war. Grund für den Anstieg der Kurven von Null an sind die 15 Sekunden in denen jeder Knoten zunächst als eigener Clusterhead arbeitete und nicht schlafen konnte. Deutlich sichtbar ist auch die Verzögerung um weitere 30 Sekunden, nach denen der Server 4 Clusterknoten wurde und das erste Mal in die Schlafphase wechseln konnte.

Offensichtlich streben die Kurven verschiedenen Grenzwerten zu. Die Server 1 bis 4 haben nach einigen Minuten konstant 94% relative Schlafdauer, die Klienten erreichen 98%. Entsprechend sind die vier Server 6% der Gesamtzeit wach, die Klienten nur 2%.

Dieser Unterschied lässt sich leicht durch die Dauer der einzelnen Schlafphasen erklären. Auch wenn die Knoten vom Clusterhead sofort nach dem Aufwachen wieder in die Schlafphase

versetzt werden, so benötigt dieser Vorgang doch einige Millisekunden, insbesondere da ein Aufruf von $A_{Coordinator}$ um bis zu 50 Millisekunden verzögert wird. Da die Klienten fünfmal längere Schlafphasen haben, erleben sie weniger Verzögerungen als die Server. Entsprechend ist ihre Wachzeit insgesamt kürzer.

Fazit Die Simulation zeigt, dass die Wachzeit, die durch das regelmäßige Aufwachen und wieder Schlafenlegen entsteht, relativ gering ist. Werden die einzelnen Schlafphasen noch länger gewählt, so werden ungenutzte Knoten nahezu die ganze Zeit schlafen.

In der Simulation wurde die Kommunikation für das Clustermanagement nicht berücksichtigt. Offensichtlich wird sie einen erheblichen Einfluss darauf haben, wie viel Energie ungenutzte Knoten sparen können.

8.2 Analyse der Scheduling-Eigenschaften

Für die zweite Simulation wurde wieder die Testumgebung mit sieben Knoten verwendet.

Ziel der Simulation In dieser Simulation sollten die Scheduling-Eigenschaften des Algorithmus' $A_{Coordinator}$ analysiert werden. Der Clusterhead sollte zum Ersten sich selber sinnvoll auslasten und viele Anfragen selber beantworten und zum Zweiten überzählige Anfragen gleichmäßig an die anderen Dienstknoten verteilen.

Durchführung der Simulation In dieser Simulation rief jeder Klient – anders als in der ersten Simulation – ständig sequentiell den entfernten `ReverseString`-Dienst auf. Zwischen dem Empfang einer Antwort und dem Senden einer neuen Anfrage pausierte jeder Klient eine zufällige Zeit, die zwischen 0,5 und 1,5 Sekunden gleichverteilt gewählt wurde.

Die Server stellten die Bearbeitungszeiten der Anfragen durch die `sleep`-Anweisung auf zufällig gewählte Werte, gleichverteilt zwischen 0,5 und 3,5 Sekunden, ein. Entsprechend dauerten die Schlafphasen der Server je rund $2\,\overline{T}_B = 4$ Sekunden.

Von Anfang an wurde ein großer Cluster mit Server 5 als Clusterhead gebildet. Auch die Klienten 1 und 2 waren Teil des Clusters.

Nach einstündiger Simulation wurde ermittelt, wie viele Anfragen jeder Server bearbeitet hatte und daraus für jeden Knoten die Wachzeit wegen Dienstnutzungen berechnet. Abbildung 19 stellt diese Wachzeit, die gemessene Schlafdauer und die Differenz zwischen gemessener und berechneter Wachzeit – die zusätzliche Wachzeit – dar.

Die Simulation wurde zweimal durchgeführt. Bei der ersten Durchführung wurde die Diensterkennung vor *jeder* Anfrage vorgenommen, bei der zweiten Durchführung nur einmal, zu Beginn der Simulation. Die Messwerte beider Durchführungen sind bei drei der sieben Knoten identisch und unterscheiden sich bei den anderen vier Knoten um weniger als ein Prozent. Dies zeigt zum Ersten, dass die Messwerte stabil sind und zum Zweiten, dass die Diensterkennung in der Simulation, im Vergleich zu den anderen Aktivitäten, in Nullzeit abläuft und daher weder sinnvoll gemessen noch bewertet werden kann.

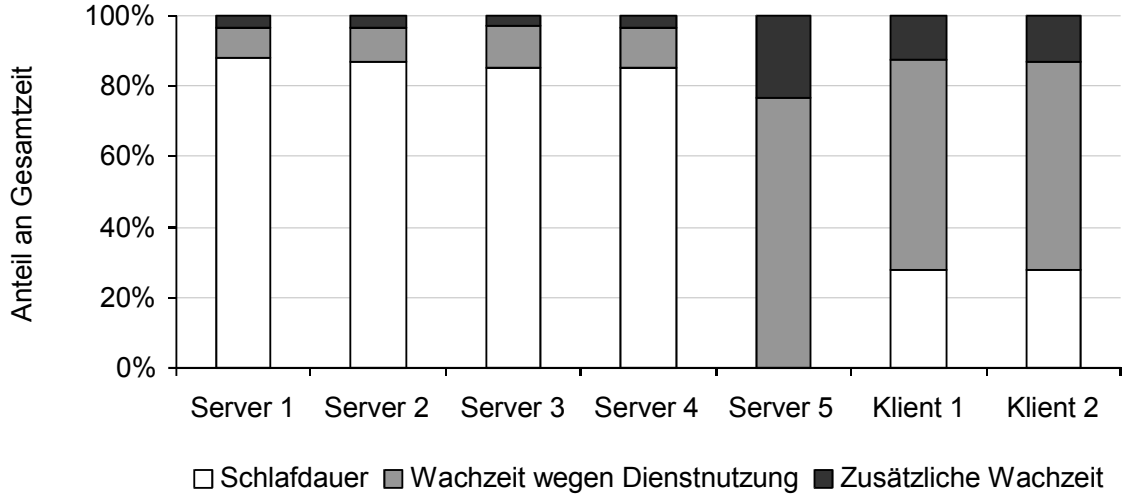

Abbildung 19: Schlafdauer und Wachzeiten bei sieben Knoten mit Dienstnutzung

Ergebnisse Abbildung 19 zeigt, dass Server 5, der Clusterhead, viele Anfragen selber bearbeitet und nicht weitergeleitet hat. Tatsächlich sind es 65% der über 2000 Anfragen. Die restlichen Anfragen hat er relativ gleichmäßig auf die vier anderen Server verteilt.

Wie oben bereits genannt, dauert es einige Millisekunden bis $A_{Coordinator}$ einen Knoten, der momentan nicht verwendet wird, tatsächlich schlafenlegt. Diese Verzögerungen verursachen eine Differenz zwischen der gemessenen und der für die Dienstnutzung berechneten Wachzeit. Bei den Servern 1 bis 4 beträgt diese Differenz rund 3% der Gesamtzeit und passt zu den in der vorherigen Simulation ermittelten relativen Wachzeiten von 6 beziehungsweise 2%.

Bei den Klienten macht diese Differenz 13% der Gesamtzeit aus, da sie viel häufiger als die Dienstknoten tatsächlich zwischen dem Schlaf- und dem Wachzustand wechseln.

Fazit Die Simulation zeigt, dass es dem hier verwendeten Algorithmus $A_{Coordinator}$ sehr gut gelingt, den Clusterhead sinnvoll auszulasten und weitere Anfragen gleichmäßig an andere Knoten zu verteilen.

8.3 Wechselnde Cluster- und Dienstkonfigurationen

Für die dritte und letzte Simulation wurde eine Testumgebung aus neun Knoten gewählt. In dieser ist jeder Knoten Klient und Dienstgeber zugleich. Ein Knoten kann bis zu drei verschiedene Dienste anbieten und nutzen. Hinter jedem der drei Dienste verbirgt sich eine Kopie der `ReverseString`-Implementierung.

Ziel der Simulation In dieser Simulation sollte getestet werden, wie gut LATE RISER mit häufigen Änderungen in der Cluster- und Dienstkonfiguration, bei gleichzeitig hoher Auslastung durch Dienstnutzungen, zurecht kommt. Die Knoten sollten, trotz der vielen Änderungen,

regelmäßig schlafen und die LATE RISER-Implementierung sollte fehlerfrei arbeiten.

Durchführung der Simulation Die Simulation wurde eine Stunde lang durchgeführt. Jeder Knoten rief ständig sequentiell entfernte Dienste auf, pausierte aber zwischen dem Empfang einer Antwort und einer neuen Anfrage gleichverteilt zwischen ein und neun Sekunden.

Des Weiteren änderte jeder Knoten alle 30 bis 90 Sekunden die Verfügbarkeit einer seiner drei Dienste, indem er die entsprechende Dienstinstanz beim Clusterhead an- oder abmeldete.

Die Cluster änderten sich nach 90 bis 150 Sekunden. Es standen vier Kombinationen zur Auswahl, von denen je eine zufällig gewählt wurde. Die Kombinationen unterschieden sich stark in Anzahl und Größe der Cluster. Allerdings war jeder der neun Knoten in nur einer Kombination Clusterhead. Da die Kombinationen gleichverteilt gewählt wurden, waren die Knoten in etwa gleich oft und lang als Clusterhead aktiv.

Die Bearbeitungszeiten der Anfragen wurden durch die `sleep`-Anweisung auf zufällig gewählte Werte, gleichverteilt zwischen 0,5 und 3,5 Sekunden, eingestellt. Entsprechend dauerten die Schlafphasen der Server je rund $2\overline{T}_B = 4$ Sekunden, wie in der anderen Testumgebung.

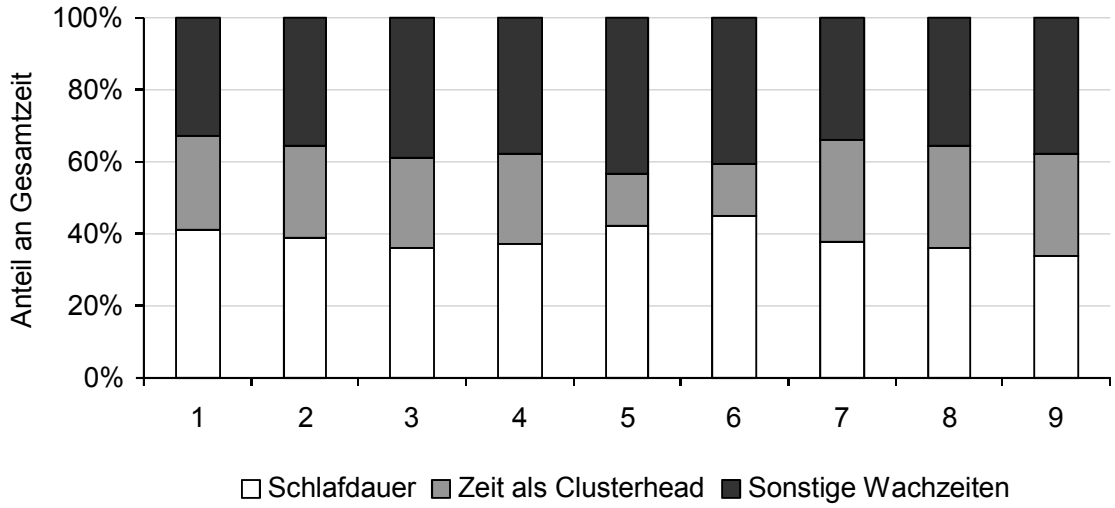

Abbildung 20: Zeiten für Dienstnutzung und -erbringung und die Aufgabe des Clusterheads

Ergebnisse In der Simulation zeigte die LATE RISER-Implementierung das gewünschte fehlerfreie Verhalten.

Abbildung 20 stellt die neun Knoten mit ihren Nummern dar und zeigt, wie lange jeder Knoten in der Summe mit den verschiedenen Aktivitäten beschäftigt war. 15 bis 28% der gesamten Zeit diente jeder Knoten als Clusterhead.

Rund 40% der Gesamtzeit schlief jeder Knoten. Die restliche Zeit waren die Knoten, als Clusterknoten, wegen Dienstnutzungen und -erbringungen wach. Auch das ständige An- und Abmelden von Dienstinstanzen, vor allem bei einem Wechsel des Clusterheads, gehört zu diesen Wachzeiten.

Fazit Die Implementierung hat sich effizient auf die ständig wechselnden Situationen eingestellt und die Knoten gleichmäßig belastet und schlafen gelegt.

9 Bewertung und Ausblick

In diesem Kapitel werden zunächst die Möglichkeiten energieeffizienter Dienstnutzung in einer Variante oder Erweiterung von SANDMAN und das zu Forschungszwecken entwickelte Verfahren LATE RISER bewertet. Die folgende Gesamtbewertung steht im Licht der in Kapitel 4.4 und Kapitel 8 vorgenommen Bewertungen und fällt entsprechend kürzer aus.

Anschließend werden Möglichkeiten zur Weiterentwicklung von LATE RISER aufgezeigt und der Blick auf offene Fragen und weitere Forschungsthemen gerichtet. Die Arbeit schließt mit einem persönlichen Fazit.

9.1 Bewertung

Die LATE RISER-Implementierung hat gezeigt, dass energieeffiziente Dienstnutzung unter Verwendung des Clusterheads als Dienststellvertreter prinzipiell möglich und sinnvoll ist. LATE RISER erfüllt viele der in Kapitel 4.2 genannten Vorteile eines solchen Vorgehens:

- *Senden von Anfragen ist einfacher:* Ein Klient kann mit nur zwei Anweisungen einen entfernten Dienst nutzen und benötigt kein Wissen darüber, wie der Clusterhead seine Anfrage verarbeitet. Auch der bei SANDMAN explizit benötigte Backoff-Mechanismus ist bei LATE RISER überflüssig.

- *Beliebige Schlaf- und Wachzeiten:* LATE RISER kann die Schlafphasen der Knoten beliebig wählen. Dass der dafür verantwortliche Algorithmus $A_{Coordinator}$ beliebig ausgetauscht werden kann, verdeutlicht dies besonders.

- *Dynamische Wahl des Dienstknotens:* Auch dieser Vorteil wird von der vorliegenden LATE RISER-Implementierung voll ausgeschöpft. $A_{Coordinator}$ kann für jede Anfrage aus allen Knoten, die eine passende Dienstinstanz anbieten, einen beliebigen wählen.

- *Einfache Netztopologie:* Die einfache Netztopologie wird bei LATE RISER bereits in der Erkennung anderer Knoten genutzt. Wie in Kapitel 6.2 erläutert, braucht ein Clusterknoten nur die Clusterheads unter den anderen Knoten zu erkennen.

Die wichtigste Eigenschaft eines Verfahrens zur energieeffizienten Dienstnutzung – die Energieersparnis – wurde in den in Kapitel 8 beschriebenen Simulationen an Hand der Schlafzeiten ermittelt und wurde festgestellt, dass LATE RISER auch dieser Anforderung genügt. Das Verfahren trägt seinen Namen zurecht, ist ein Knoten im Leerlauf bei entsprechend langen Schlafphasen doch weniger als 2% der Gesamtzeit wach.

Die Energieersparnis durch Schlafphasen funktioniert aber nicht nur im Leerlauf hervorragend, sondern hat auch in den verschiedenen Testumgebungen bei unterschiedlichsten Anfrageraten beachtliche Ergebnisse erbracht. Eine gute Anpassung an gegebene Situationen ist

zum Großteil Verdienst von $A_{Coordinator}$. Umso erstaunlicher ist, dass der hier relativ einfach gewählte Algorithmus $A_{Coordinator}$ bereits so gute Ergebnisse erzielt.

Leider kann von den Simulationsergebnissen mit LATE RISER nicht Eins-zu-eins auf die Leistungsfähigkeit eines ähnlichen Verfahrens für beliebige ubiquitäre Rechnersysteme geschlossen werden, obwohl LATE RISER in BASE – einer *realen* Middleware für ubiquitäre Rechnersysteme – und nicht einer reinen Simulationssoftware implementiert wurde. Der Grund ist, dass mit BASE bisher nur Dienste verwendet werden können, die LATE RISER ebenfalls unterstützt. Wie in Kapitel 6.4 erläutert, sind dies ausschließlich Dienste, die dem Anfrage-Antwort-Muster folgen. Die in Kapitel 5.6 diskutierten Arbeiten [Chandra 2002, ChaVah 2002, KraKri 2000] und die Klassifikation von Diensten aus [KoObKl 2004, AP. 1.2] zeigen, dass ein Verfahren zur energieeffizienten Dienstnutzung mit noch ganz anderen Kommunikationsmustern zurecht kommen sollte.

Die in Kapitel 4.3 aufgezeigten Nachteile einer energieeffizienten Dienstnutzung mit dem Clusterhead als Dienststellvertreter haben zu weiteren Einschränkungen geführt, die bei der Entwicklung von LATE RISER berücksichtigt wurden und auch von anderen ähnlichen Verfahren zu beachten sind. Zu den wichtigen, schwer oder gar nicht kompensierbaren Nachteilen, gehören unter anderem das erhöhte Nachrichtenaufkommen, die aus Sicht des Klienten unregelmäßigen Bearbeitungszeiten, die neuen Fehlerquellen durch einen Ausfall des Clusterheads und der aufwändige Wechsel desselben.

Entsprechend wurden in Kapitel 4.4 zwei Bedingungen formuliert, die erfüllt sein müssen, soll eine energieeffiziente Dienstnutzung mit dem Clusterhead als Dienststellvertreter sinnvoll eingesetzt werden. Die Erkenntnisse aus der Entwicklung und Implementierung von LATE RISER unterstreichen diese Bedingungen, anstatt sie zu relativieren:

1. Die Mehrzahl der Dienste muss ohne oder mit nur geringem Zusatzaufwand mit dem Clusterhead als Dienststellvertreter verwendet werden können.

2. Ein durchschnittlicher Knoten muss die Aufgabe des Clusterheads übernehmen können, ohne wegen zu geringer Rechenleistung oder Speicherkapazität zum Engpass zu werden.

Es bleibt festzustellen: Dienstnutzung mit dem Clusterhead als Dienststellvertreter bietet in spontan vernetzten ubiquitären Rechnersystemen prinzipiell ein großes Energiesparpotential, kann allerdings nicht in allen Systemen wirklich sinnvoll eingesetzt werden.

9.2 Weiterentwicklung von LATE RISER

LATE RISER wurde in dieser Arbeit zu Forschungszwecken entwickelt und implementiert. In drei Bereichen lässt sich LATE RISER weiterentwickeln und zu neuen Simulationen verwenden.

Implementierung In der Implementierung wurde im Rahmen dieser Arbeit auf ein Sessionkonzept verzichtet, obwohl es für viele Dienste ein Muss ist. Für den Vergleich mit SANDMAN ist es unverzichtbar.

Gleiches gilt für die in Kapitel 4.3 und Kapitel 6.5 ausführlich diskutierten Folgeanfragen. Die LATE RISER-Implementierung muss sie in geeigneter Form unterstützen, sollen Anwendungen, die direkt hintereinander viele Anfragen stellen, nicht ernsthafte Leistungsprobleme bekommen.

Analyse Nächstes Ziel bei der Analyse sollte ein expliziter Vergleich mit SANDMAN sein, da LATE RISER eine Variante beziehungsweise Erweiterung dieses Verfahrens darstellt. Für einen sinnvollen Vergleich müssen neben dem Energieverbrauch der Knoten auch Aspekte wie Antwortzeiten und Kollisionen berücksichtigt werden.

Für eine möglichst realitätsnahe Analyse muss LATE RISER unter folgenden drei Bedingungen getestet werden:

1. *Reales Clustermanagement:* Das Clustermanagement wird nicht simuliert, sondern erfolgt entsprechend einem bewährten Algorithmus zur Clusterbildung. Der Clusterhead wird nicht vorab bestimmt, sondern an Hand von Gesichtspunkten wie Energievorrat und Position in der Netztopologie gewählt.

2. *Beliebige Knotenbewegungen und -verfügbarkeit:* Knoten wechseln nicht nur die Cluster, sondern bewegen sich zufällig in die Reichweite bestimmter Knoten und umgekehrt. Ein Knoten kann schlagartig ausfallen oder neu zum System hinzukommen.

   Soll die Analyse durch Simulation erfolgen, sind hierfür geeignete Mobilitäts- und Aktivitätsmodelle erforderlich.

3. *Große Auswahl an Diensten:* Es stehen deutlich mehr und sehr unterschiedliche Dienste zur Verfügung. Die Dienste unterscheiden sich nicht nur in der Bearbeitungszeit von Anfragen, sondern auch stark in der benötigten Rechenleistung und Speicherkapazität.

   In der Simulation sollte sich die Auswahl an Diensten an existierenden Szenarien orientieren und die Klienten die Dienste entsprechend realer Anwendungsprofile nutzen.

Koordinierung der Anfragen und Schlafphasen Auch $A_{Coordinator}$ lässt sich weiterentwickeln oder durch einen völlig neuen Algorithmus ersetzen.

Ein künftiger Algorithmus sollte Knoten, die gleiche Dienste anbieten, abwechselnd schlafenlegen. In der hier beschriebenen Implementierung nutzt $A_{Coordinator}$ diese Möglichkeit nicht explizit, obwohl sie Verzögerungen bei der Dienstnutzung völlig eliminieren kann.

9.3 Weitere Forschungsthemen

Viele der in Kapitel 4.2 genannten Möglichkeiten bei der energieeffizienten Dienstnutzung mit dem Clusterhead als Dienststellvertreter werden von LATE RISER nicht genutzt. Dazu gehören das Schlafenlegen von Knoten, die momentan als Klient einen Dienst nutzen, Fehlertoleranz durch geeignetes Verhalten des Clusterheads und einfacher Zugriff auf zusammengesetzte Dienste. Besonders das Schlafenlegen von Klienten wirft eine Reihe von Fragen auf, die separat diskutiert werden müssen:

- Wie ist die Dringlichkeit einer Antwort gegen die Verzögerung durch Schlafphasen des Klienten zu bewerten?
- Wie soll ein Clusterhead mit Klienten umgehen, die nicht Teil seines Clusters sind?
- Wie kann mit Klienten umgegangen werden, die verschiedene Dienste gleichzeitig nutzen?

Wahl der Clusterheads Die Wahl des Clusterheads wird bei LATE RISER vollständig einem beliebigen Clustermanagement-Algorithmus überlassen. Aus dem Routing existieren geeignete Algorithmen, wie in Kapitel 5.4 diskutiert. Diese berücksichtigten bei der Wahl des Clusterheads aber weder die von den Knoten angebotenen Dienste noch besondere Knoteneigenschaften wie Rechenleistung oder Speicherkapazität. In diesem Zusammenhang sind zwei Fragen zu beantworten:

- Welche Vor- und Nachteile ergeben sich, werden derartige Knoteneigenschaften bei der Wahl des Clusterheads berücksichtigt?
- Wie können solche Knoteneigenschaften sinnvoll bewertet werden – sind sie doch schwer in Zahlen zu fassen?

Kombination drahtloser Netze Rechner ubiquitärer Systeme haben oft Schnittstellen zu verschiedenen drahtlosen Netzen. Anscheinend völlig unerforscht ist bisher, wie diese Netze kombiniert werden können. Könnte eine Nachricht dank des Clusterheads die Netze wechseln, so kann ein Knoten, der beispielsweise nur über eine Funknetzschnittstelle verfügt, einen Dienst nutzen, der von einem Knoten angeboten wird, welcher nur über eine Infrarotschnittstelle erreichbar ist. Es bleibt zu klären:

- Wie können verschiedene drahtlose Netze technisch kombiniert werden?
- Welche Möglichkeiten und Probleme ergeben sich im Besonderen für die energieeffiziente Nutzung von Diensten?

Bewertung energieeffizienter Dienstnutzung Energieeffiziente Dienstnutzung mit dem Clusterhead als Dienststellvertreter ist nicht immer sinnvoll. Zwei Beispiele sind:

1. *Ungeeignete Dienste:* Wie oben genannt, sind manche Dienste – die zum Beispiel Videodatenströme verwenden – für die Nutzung über den Clusterhead sehr ungeeignet. Oft existieren für solche Dienste aber andere Verfahren zur energieeffizienten Dienstnutzung.
2. *Sehr aktive Knoten:* Sind die meisten oder alle Knoten eines ubiquitären Rechnersystems ständig aktiv und können gar nicht in die Schlafphase wechseln, so kostet der Umweg über den Clusterhead nicht nur unnötig Energie, sondern macht sich auch in der Halbierung der Bandbreite des Netzes sehr unangenehm bemerkbar, da jede Nachricht das Übertragungsmedium doppelt beansprucht.

Ein ubiquitäres Rechnersystem sollte entsprechend an Hand von Energieverbrauch, Knotenmobilität und -verfügbarkeit, Auswahl an Diensten, Rechnereigenschaften, Kommunikationsmuster entscheiden, ob und welches Verfahren zur energieeffizienten Dienstnutzung im Moment geeignet ist. Dazu ist zu erforschen:

- Wie sind die genannten Aspekte – vor allem die Dienste – zu klassifizieren und zu bewerten?
- Wie kann die Bewertung dieser Aspekte mit *geringem* Aufwand zur Laufzeit eines ubiquitären Rechnersystems erfolgen?

Ausblick Energieeffiziente Dienstnutzung unter Einsatz von Anwendungswissen – ob mit oder ohne Clusterhead als Dienststellvertreter – ist ein wichtiges Forschungsgebiet, welches noch viele ungeahnte Möglichkeiten bietet, ohne die typischen Nachteile existierender energieeffizienter Algorithmen, sinnvoll Energie zu sparen.

9.4 Persönliches Fazit

Mit dem Ablauf dieser Arbeit bin ich sehr zufrieden, auch wenn ich zwischendurch mehrfach große und zeitintensive Korrekturen vornehmen musste. Das Vorgehen, zunächst eine Studienarbeit wie diese und erst danach die Diplomarbeit zu verfassen, halte ich für ein wirklich erfolgreiches Absolvieren des Studiums für unverzichtbar.

Ich sehe die Studienarbeit als hervorragende Übung der wissenschaftlichen Arbeitsweise und danke meinem Betreuer Gregor Schiele für alle Unterstützung und Anregung. Mein Dank gilt auch seinem Kollegen Marcus Handte, der mich ausführlich in BASE eingeführt hat und allen Anderen, die mir mit Ratschlägen zur Seite gestanden sind.

Literatur

[3PC] Kurt Rothermel; Christian Becker; Gregor Schiele u.a.: *Peer to Peer Pervasive Computing*. `http://www.3pc.info`. Zugriffsdatum: 3. Januar 2005.

[Angstm 2003] Karsten Angstmann: *Entwicklung eines energieeffizienten Verfahrens zur dynamischen Erkennung entfernter Dienste in ubiquitären Informationssystemen*. Universität Stuttgart [Diplomarbeit]. November 2003.

[Atheros 2003] Atheros Communications, Inc.: *Power Consumption and Energy Efficiency Comparisons of WLAN Products*. `http://www.atheros.com/pt/papers.html`. Mai 2003. Zugriffsdatum: 2. August 2004.

[BeJa+ 2001] Benjie Chen; Kyle Jamieson; Hari Balakrishnan; Robert Morris: *Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad hoc Wirless Networks*. In: Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), S. 85–96. Rom, Italien. Juli 2001.

[BluSIG 2003] Bluetooth Special Interest Group (SIG): *Bluetooth Core Specification v1.2*. `http://www.bluetooth.org/`. November 2003. Zugriffsdatum: 15. Oktober 2004.

[Chandra 2002] Surendar Chandra: *Wireless Network Interface Energy Consumption Implications of Popular Streaming Formats*. In: Martin Kienzle und Prashant Shenoy (Hgg.). Multimedia Computing and Networking (MMCN), Vol. 4673, S. 85–99. San Jose, USA. Januar 2002.

[ChaVah 2002] Surendar Chandra; Amin Vahdat: *Application-specific network management for energy-aware streaming of popular multimedia formats*. USENIX Annual Technical Conference. Monterey, USA. Juni 2002.

[DamLan 2003] Tijs van Dam; Koen Langendoen: *An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks*. In: Proceedings of the first international Conference on Embedded Networked Sensor Systems (SenSys), S. 171–180. Los Angeles, USA. November 2003.

[EunVai 2002] Eun-Sun Jung; Nitin H. Vaidya: *An Energy Efficient MAC Protocol for Wireless LANs*. The 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM). New York, USA. Juni 2002.

[FeeNil 2001] Laura Marie Feeney; Martin Nilsson: *Investigating the Energy Consumption of a Wireless Network Interface in an Ad Hoc Networking Environment*. In: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM). Anchorage, USA. April 2001.

[Gilhooly 2004] Kevin Gilhooly: *Using WebSphere Studio Device Developer to Build Embedded Java Applications*. IBM Developer Technical Support Center. Dallas, USA. `http://www.redbooks.ibm.com/abstracts/SG247082.html`. April 2004. Zugriffsdatum: 8. Dezember 2004.

[IEEE 802.11] IEEE Computer Society LAN MAN Standards Committe (Hg.): *IEEE, Std 802.11: Wireless LAN Medium Access Control and Physical Layer Specications*. IEEE Standards Association. `http://standards.ieee.org/getieee802/`. 1999. Zugriffsdatum: 2. August 2004.

[IkiOga 2003] Selma Ikiz; Vinit [A.] Ogale: *Coordinated Energy Conservation for Ad Hoc Networks*. `http://www.ece.utexas.edu/~ogale/energy_conservation.pdf`. Universität von Texas. Austin, USA. August 2003. Zugriffsdatum: 15. Oktober 2004.

[IrDA 1996] Infrared Data Association: *Link Management Protocol*. `http://www.irda.org/`. Januar 1996. Zugriffsdatum: 15. Oktober 2004.

[Irmscher 2004] Klaus Irmscher: *Verteilte Systeme*. Universität Leipzig [Scriptum zur gleichnamigen Vorlesung]. Juli 2004.

[J2ME] Sun Microsystems: *Java 2 Platform, Micro Edition*. `http://java.sun.com/j2me/`. Zugriffsdatum: 26. Oktober 2004.

[JoSi+ 2001] Christine E. Jones; Krishna E. Sivalingam; Prathima Agrawal; Jyh-Cheng Chen: *A Survey of Energy Efficient Network Protocols for Wireless Networks*. In: Wireless Networks, Vol. 7, No. 4, S. 343–358. Juli 2001.

[Karn 1990] Phil Karn: *MACA – A New Channel Access Method for Packet Radio*. In: ARRL/CRRL Amatuer Radio 9th Computer Networking Conference, S. 134–140. London, Kanada. September 1990. `http://www.ka9q.net/papers/`. Zugriffsdatum: 20. Oktober 2004.

[Kirste 2002] Thomas Kirste: *Mobile Informationssysteme und Ubiquitous Computing*. Technische Universität Darmstadt [Folien zur Vorlesung Context-Awareness, Situierte Interaktion und Pervasive Computing]. Oktober 2002.

[KlKoOb 2003] Michael Klein; Birgitta König-Ries; Philipp Obreiter: *Lanes – A Lightweight Overlay for Service Discovery in Mobile Ad Hoc Networks*. In: Second International Workshop on Wireless Information Systems (WIS), No. 6/2003. Oder (hier verwendet): `http://wwwipd.ira.uka.de/DIANE/`. 2003. Zugriffsdatum: 2. August 2004.

[KoObKl 2004] Birgitta König-Ries; Philipp Obreiter; Michael Klein: *Effektive und effiziente Dienstsuche und -nutzung in Ad-hoc-Netzen*. Universität Karlsruhe [Zwischenbericht]. `http://wwwipd.ira.uka.de/DIANE/`. Februar 2004. Zugriffsdatum: 2. August 2004.

[KraBal 2002] Ronny Krashinsky; Hari Balakrishnan: *Minimizing Energy for Wireless Web Access with Bounded Slowdown*. In: Proceedings of the Eighth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), S. 119–130. Atlanta, USA. September 2002.

[KraKri 2000] Robin Kravets and P. Krishnan: *Application-driven power management for mobile communication*. In: Wireless Networks, Vol. 6, No. 4, S. 263–277. Juli 2000.

[MatLan 2003] Friedemann Mattern; Marc Langheinrich: *Die Informatisierung des Alltags*. In: Bulletin ETH Zürich Nr. 291, S. 15–18. November 2003.

[Mattern 2003] Friedemann Mattern: *Vom Verschwinden des Computers – die Vision des Ubiquitous Computing*. In: Total vernetzt – Szenarien einer informatisierten Welt, S. 1–41. Springer-Verlag. April 2003.

[Nidd 2001] Michael Nidd: *Service Discovery in DEAPspace*. In: IEEE Personal Communications, Vol. 8, No. 4, S. 39–45. August 2001.

[NS2] Information Sciences Institute (ISI) u.a.: *The Network Simulator – ns-2*. `http://www.isi.edu/nsnam/ns/`. Zugriffsdatum: 3. Januar 2005.

[PatNag 2001] Dipesh Patel; Rakesh Nagi: *Ad hoc Wireless Networks*. Universität Buffalo [Präsentation] `http://www.acsu.buffalo.edu/~nagi/courses/684/adhoc.pdf`. Oktober 2001. Zugriffsdatum: 11. November 2004.

[PeBeDa 2003] Charles E. Perkins; Elizabeth M. Belding-Royer; Samir Das: *Ad Hoc On Demand Distance Vector Routing*. IETF RFC 3561. `http://moment.cs.ucsb.edu/pub/rfc3561.txt`. Juli 2003. Zugriffsdatum: 20. Oktober 2004.

[Pepper 1995] Peter Pepper: *Grundlagen der Informatik*. Olderbourg Verlag. München. Mai 1995.

[Perkins 2000] Charles E. Perkins: *Ad Hoc Networking*. Addison-Wesley. Dezember 2000.

[ScBeRo 2004] Gregor Schiele; Christian Becker; Kurt Rothermel: *Energy-Efficient Cluster-based Service Discovery for Ubiquitous Computing*. In: Proceedings of the 11th ACM SIGOPS European Workshop. Leuven, Belgium. September 2004.

[SiBe+ 2000] Tajana Simunic; Luca Benini; Peter Glynn; Giovanni De Micheli: *Dynamic Power Management for Portable Systems*. In: Proceedings of the Sixth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), S. 11–19. Boston, USA. August 2000.

[SinRag 1998] Suresh Singh; C[auligi] S. Raghavendra: *PAMAS – Power Aware Multi-Access protocol with Signalling for Ad Hoc Networks*. In: ACM Computer Communication Review, Vol. 28, No. 3, S. 5–26. Juli 1998.

[SiWoRa 1998] Suresh Singh; Mike Woo; C[auligi] S. Raghavendra: *Power-Aware Routing in Mobile Ad Hoc Networks*. In: In Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), S. 181–190. Dallas, USA. Oktober 1998.

[SriChi 2002] Chavalit Srisathapornphat; Chien-Chung Shen: *Coordinated Power Conservation for Ad hoc Networks*. In: IEEE International Conference on Communications (ICC), Vol. 5, S. 3330–3335. New York, USA. April 2002.

[StGa+ 1996] Mark Stemm; Paul Gauthier; Daishi Harada; Randy H. Katz: *Reducing Power Consumption of Network Interfaces in Hand-Held Devices*. In: Proceedings of the 3rd International Workshop on Mobile Multimedia Communications (MoMuC). Princeton, USA. September 1996.

[TechTa 2003] TechTarget: *Glossary Definition: Ad-hoc Network*. `http://searchmobilecomputing.techtarget.com/gDefinition/`. April 2003. Zugriffsdatum: 11. November 2004.

[WeHeEs 2002] Wei Ye; John Heidemann; Deborah Estrin: *An Energy-Efficient MAC Protocol for Wireless Sensor Networks*. In: Proceedings of the 21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), Vol. 3, S. 1567–1576. New York, USA. Juni 2002.

[Weiser 1991] Mark Weiser: *The Computer for the 21st Century*. In: Scientific American, Vol. 265, No. 3, S. 94–104. September 1991.

[WSDD] International Business Machines Corporation: *WebSphere Studio Device Developer 5.7.1*. `http://www-306.ibm.com/software/wireless/wsdd/`. 2004. Zugriffsdatum: 3. Januar 2005.

[YaBi+ 2003] Ya Xu; Solomon Bien; Yutaka Mori; John Heidemann; Deborah Estrin: *Topology Control Protocols to Conserve Energy in Wireless Ad Hoc Networks*. Center for Embedded Networked Sensing (CENS), University of California: Technical Report #6. `http://www.cens.ucla.edu/portal/people/estrin.html`. January 2003. Zugriffsdatum: 15. Oktober 2004.

[YaHeEs 2000] Ya Xu; John Heidemann; Deborah Estrin: *Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks*. Information Sciences Institute (ISI), University of Southern California (USC): Research Report 527. Marina del Rey, USA. Oktober 2000. `http://www.isi.edu/~johnh/PAPERS/Xu00a.html`. Zugriffsdatum: 15. Oktober 2004.

[YaHeEs 2001] Ya Xu; John Heidemann; Deborah Estrin: *Geography-informed Energy Conservation for Ad Hoc Routing*. In: Proceedings of the Seventh Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), S. 70–84. Rom, Italien. Juli 2001.

[ZanPel 2004] Andrea Zanella; Francesco De Pellegrini: *Mathematical Analysis of IEEE 802.11 Energy Efficiency*. In: Proceedings of the Seventh International Symposium on Wireless Personal Multimedia Communications (WPMC). Abano Terme, Italien. September 2004.

[Zitter 2004] Martina Zitterbart: *Informatik II*. Universität Karlsruhe [Foliensatz zur gleichnamigen Vorlesung]. April 2004.

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.