System Support for Spontaneous Pervasive Computing Environments

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart zur Erlangung der
Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

vorgelegt von

Gregor Alexander Schiele

aus Stuttgart

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. phil. nat. Christian Becker
Tag der mündlichen Prüfung:	27.04.2007

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2007

Contents

Abstract 17

Deutsche Zusammenfassung 19

1 Introduction 29

1.1 Motivation . . . 29
1.2 Classification . . . 32
1.3 Contributions . . . 34
1.4 Structure . . . 35

2 Smart Peer Groups 37

2.1 The Smart Peer Group Model . . . 37
2.2 Main Challenges . . . 39
2.2.1 Heterogeneity . . . 39
2.2.2 Dynamism . . . 40
2.2.3 Resource Constraintness . . . 40
2.3 Requirements . . . 41
2.3.1 Dynamic Device Lookup . . . 41
2.3.2 Adaptive and Interoperable Communication Support . . . 41
2.3.3 Decentralized Operation . . . 41
2.3.4 Configurability . . . 42
2.3.5 Energy-efficiency . . . 42
2.4 System Model . . . 42
2.4.1 Device Model . . . 42
2.4.2 Network Model . . . 44
2.4.3 User Model . . . 44
2.5 Summary . . . 45

3 A Communication Middleware for Smart Peer Groups 47

3.1 Design Rationale . . . 47
3.1.1 Uniform Programming Interface . . . 48

3.1.2 Communication Model Decoupling 48
3.1.3 Strategy-based Communication Adaptation 49
3.1.4 Micro-broker-based Architecture 49
3.1.5 Device Deactivation 50
3.2 Architecture 52
3.2.1 Invocations 54
3.2.2 Stubs and Skeletons 54
3.2.3 Communication Models 55
3.2.4 InvocationBroker 55
3.2.5 Service and Device Registry 57
3.2.6 PluginManager 57
3.2.7 Plugin Selection Strategy 58
3.3 Implementation 60
3.4 Related Work 61
3.4.1 Conventional Middleware Systems 61
3.4.2 Dynamically Reconfigurable Middleware 61
3.4.3 Middleware for Resource-Poor Devices 62
3.4.4 Summary 62
3.5 Conclusion 62

4 A Service Discovery System for Smart Peer Groups 63

4.1 Discovery Objectives 63
4.1.1 Discovery Latency (l) 64
4.1.2 Discovery Precision (p) 64
4.1.3 Discovery Recall (r) 65
4.1.4 Discussion 66
4.2 Design Rationale 66
4.2.1 System Organization 66
4.2.2 Information Distribution Pattern 69
4.3 SANDMAN 70
4.3.1 System Structure 71
4.3.2 Node Roles 72
4.3.3 Basic Communication Primitives 72
4.4 Related Work 74
4.4.1 Conventional Service Discovery Systems 74
4.4.2 Service Discovery for Resource Poor Devices 75
4.4.3 Energy-efficient Service Discovery 75
4.4.4 Energy-efficient Communication 76
4.4.5 Summary 77

4.5 Conclusion 77

5 Cluster Management 79

5.1 Requirements 79
5.2 Design Rationale 80
5.2.1 Group Mobility 81
5.2.2 Topology Management 82
5.2.3 Neighborhood Detection 83
5.2.4 Cluster Formation and Join 83
5.2.5 Cluster Head Election 85
5.2.6 Cluster Membership 88
5.2.7 Periodic Probing 88
5.2.8 Cluster Handover 90
5.3 Data Structures and Variables 91
5.3.1 Data Structures 91
5.3.2 Variables 92
5.4 Algorithms: Neighborhood Detection and Cluster Preparation 93
5.4.1 Announcements 93
5.4.2 Detecting Joinable Nodes 95
5.5 Algorithms: Cluster Creation and Join 95
5.6 Algorithms: Cluster Maintenance 98
5.6.1 Lease Renewal 98
5.6.2 Cluster Handover 101
5.7 Handling Communication Errors 103
5.7.1 Lost Announcement Messages 103
5.7.2 Lost LeaseRequest Messages 104
5.8 Cluster Management in a Smart Environment 105
5.9 Services Provided to Other Subsystems 105
5.9.1 Services Provided to Energy Management 106
5.10 Related Work 107
5.10.1 Stability-aware Approaches 108
5.10.2 Energy-efficient Clustering 108
5.10.3 Summary 110
5.11 Conclusion 110

6 Service Management 111

6.1 Overview 111
6.2 Design Rationale 112
6.2.1 Service Availability 113
6.2.2 Discovery Range 114

6.2.3 Service Description Language 115
6.3 Data Structures 116
6.3.1 ServiceID 116
6.3.2 ServiceDescriptor 116
6.3.3 ServiceEntry 116
6.3.4 ServiceRegistry 117
6.3.5 ResultSet 118
6.4 Service Provider Interface 118
6.4.1 Registration 119
6.4.2 Update 120
6.4.3 Removal 121
6.5 Client Interface 122
6.5.1 Initiating a Lookup Request 122
6.5.2 Answering a Lookup Request 124
6.6 ServiceRegistry Maintenance 125
6.6.1 Joining a Cluster 126
6.6.2 Cluster Handover 126
6.6.3 Loosing a Cluster Member 127
6.6.4 Loosing the Cluster Head Role 127
6.6.5 Changing Plugin Availability 127
6.7 Services Provided to Other Subsystems 128
6.7.1 Services Provided to Cluster Management 128
6.7.2 Services Provided to Energy Management 128
6.8 Conclusion 130

7 Energy Management 131
7.1 Overview 131
7.2 Design Rationale 132
7.2.1 Node Deactivation 133
7.2.2 Node Reactivation 133
7.3 Isolated Approach 134
7.3.1 Deactivation Time 134
7.3.2 Deactivation Length 135
7.3.3 Deactivation Negotiation 136
7.3.4 Transition Algorithm 138
7.3.5 Discussion 144
7.4 Coordinated Approach 144
7.4.1 Transition Scheduling 145
7.4.2 Transition Algorithm 147

7.5 Conclusion 149

8 Evaluation **151**
8.1 Performance Metrics 151
8.2 System Parameters 153
8.3 Analytical Evaluation 160
8.3.1 Message Overhead 160
8.3.2 Energy Savings 162
8.3.3 Discovery Precision and Recall 167
8.3.4 Discovery Latency 171
8.3.5 System Parameterization 171
8.4 Experimental Evaluation 172
8.4.1 Emulation Environment 172
8.4.2 Experimental Settings 174
8.4.3 Experimental Results 176
8.5 Summary and Conclusion 181

9 Conclusion and Outlook **183**
9.1 Conclusion 183
9.2 Outlook 184

Bibliography **185**

List of Figures

1.1 Smart Environments . . . 30
1.2 Smart Peer Groups . . . 31
1.3 Classification . . . 34

2.1 System model . . . 43
2.2 Power State Machine of a Node . . . 43

3.1 BASE architecture . . . 53
3.2 Invocation object structure . . . 54
3.3 Request / response in BASE . . . 56
3.4 Plugin selection . . . 59

4.1 Discovery system organization . . . 67
4.2 SANDMAN approach overview . . . 70
4.3 SANDMAN structure . . . 71
4.4 SANDMAN architecture . . . 72
4.5 DEAPspace service discovery . . . 76

5.1 Cluster Formation Example . . . 84
5.2 Cluster Head Election Example . . . 87
5.3 Periodic probing scenario . . . 89
5.4 Handover scenario . . . 90

6.1 SANDMAN service management . . . 112
6.2 SANDMAN service management interactions . . . 113

7.1 SANDMAN energy management interactions . . . 132
7.2 Isolated energy management example . . . 137
7.3 Coordinated energy management example . . . 146

8.1 Message overhead for cluster formation . . . 162
8.2 Message overhead (per second) for running cluster . . . 163

8.3 Energy savings for different cluster sizes and deactivation lengths 167
8.4 Energy savings for different cluster sizes and stability levels 168
8.5 Precision for different cluster sizes and stabilities 169
8.6 Recall for different cluster sizes and stabilities 170
8.7 Recall for different cluster sizes and stabilities 171
8.8 Experimental evaluation scenario overview 174
8.9 Total message overhead per second 176
8.10 Energy savings 177
8.11 Discovery precision 178
8.12 Discovery recall 179
8.13 Discovery latency overhead for not using SANDMAN (1) 180
8.14 Discovery latency overhead for not using SANDMAN (2) 181

List of Tables

2.1 Parameters used for the energy model 44

3.1 Power consumption of an Orinoco Gold IEEE 802.11b adapter 51

4.1 Parameters used for describing service discovery objectives 65

5.1 Variables used by the cluster management 93

7.1 Parameters used by the energy management (1) 136
7.2 Parameters used by the energy management (2) 145

8.1 Mobility model parameters 154
8.2 Energy model parameters 155
8.3 Environmental parameters 156
8.4 SANDMAN parameters 158
8.5 Parameter settings for experimental evaluation 175

List of Procedures

5.1 *OnAncTimeout*: Handling an AnnouncementTimeout event 94
5.2 *OnAnc*: Receiving an ANC message 94
5.3 *IsJoinable*: Determine if a given node could be joined 95
5.4 *OnJoinReq*: Receiving a JOIN_REQ message 95
5.5 *SwitchRole*: Switch node role 96
5.6 *OnJoinAck*: Receiving a JOIN_ACK message 97
5.7 *OnJoinNAck*: Receiving a JOIN_NACK message 97
5.8 *OnLeaseRenewalTimeout*: Initiating a lease renewal 98
5.9 *OnLeaseReq*: Receiving a LEASE_REQ message 99
5.10 *OnGrantLease*: Receiving a GRANT_LEASE message 100
5.11 *OnLeaseExpired*: Handling a LeaseExpired event 101
5.12 *OnHandoverReq*: Receiving a HO_REQ message 102
5.13 *OnHandoverAck*: Receiving a HO_ACK message 102
5.14 *OnHandoverNAck*: Receiving a HO_NACK message 103
6.1 *Register*: Register a service 119
6.2 *OnRegister*: Receiving a REGISTER message 120
6.3 *Update*: Update a service registration 121
6.4 *OnUpdate*: Receiving an UPDATE message 121
6.5 *Remove*: Remove a service registration 122
6.6 *Lookup*: Search for services 123
6.7 *OnLookupReq*: Receiving a LOOKUP_REQ message 125
6.8 *LastDiscovery*: Compute last time a node’s services were discovered 129
6.9 *UpdateSchedule*: Update a node’s deactivation schedule 129
7.1 *OnIdleTimerExpired*: Called when the IdleTimer expires 139
7.2 *Deactivate*: Deactivating a node 140
7.3 *OnSleepAnc*: Receiving a SLEEP_ANC message 141
7.4 *OnSleepAckReceived*: Receiving a SLEEP_ACK message 143
7.5 *Deactivate*: Deactivation of a node 147
7.6 *OnSleepAnc*: Receiving a SLEEP_ANC message 148

List of Abbreviations

3PC	Peer-to-peer pervasive computing
CDR	Common Data Representation
CH	Cluster Head
CN	Clustered Node
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
DPM	Dynamic Power Management
GPS	Global Positioning System
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter-Orb Protocol
IP	Internet Protocol
IrDA	Infrared Data Association
LUS	Lookup Service
MAC	Medium Access Control
MANET	Mobile Ad hoc Network
MCDS	Minimum Connected Dominating Set
NET	Network Emulation Toolkit
PDA	Personal Digital Assistant
PNA	Personal Navigation Assistant
RPC	Remote Procedure Call
SANDMAN	Service Awareness and Discovery in Mobile Ad hoc Networks
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
UMTS	Universal Mobile Telecommunications System
UN	Unclustered Node
UPnP	Universal Plug and Play
XDR	External Data Representation
XML	Extensible Markup Language

Abstract

Mobile networked devices become more and more pervasive. By embedding such devices into everyday items, pervasive computing systems will emerge in the near future. Current approaches for such systems are based on the model of Smart Environments. In such environments, a preinstalled hardware infrastructure enhances a spatial area, e.g., a room or house, and enables it to coordinate multiple mobile devices present in the environment to cooperatively provide services to the users. However, such systems rely on the presence of such an expensive infrastructure and do not work in areas without it. This restricts the deployment of pervasive computing systems severely. Therefore, in this work we propose another model for pervasive computing systems, the so-called Smart Peer Group model. A Smart Peer Group consists of a number of interconnected mobile devices that discover each other dynamically and form a spontaneous composition of devices. Coordination is provided by the participating devices themselves and no external infrastructure is needed. This results in a highly flexible system that can be used at any time and anywhere. The development of such systems is a non trivial task, due to the high level of dynamism, the potentially high resource constraintness, and the unpredictable nature of Smart Peer Groups. In this dissertation, we present the Smart Peer Group model and analyze the specific characteristics of this system class. In addition, we propose a number of concepts and algorithms to develop Smart Peer Group-based Pervasive Computing systems. A communication middleware for Smart Peer Groups is presented, which offers means to cope with resource-poor specialized devices and shields application developers from fluctuating network connectivities. Furthermore, a service discovery system for such systems is developed, which allows unused devices to temporarily deactivate themselves in order to save energy without loosing the ability to discover new services or to be discovered by others. The presented concepts and algorithms are evaluated in different scenarios using an analytical and an experimental evaluation.

Deutsche Zusammenfassung

Systemunterstützung für Spontan Vernetzte Ubiquitäre Rechnersysteme

1 Einleitung

Mit der zunehmenden Verbreitung immer leistungsstärkerer und kompakterer mobiler Rechnersysteme durchdringen diese unsere alltägliche Umgebung immer mehr. Millionen von Anwendern nutzen schon heute Mobiltelefone, tragbare Navigationssysteme und Laptops. Gleichzeitig erlauben es neue Vernetzungstechnologien, Geräte mit immer größeren Übertragungsraten und geringeren Kosten miteinander zu vernetzen. Mittelfristig ist davon auszugehen, dass Anwender immer und überall von einer Vielzahl elektronischer Geräte umgeben sein werden, die mittels drahtloser Kommunikation miteinander kooperieren und Anwendern so gemeinsam Informationen und Dienstleistungen bereitstellen werden. Geräte können hierbei für den Anwender unsichtbar in Gegenstände des täglichen Bedarfs eingebettet oder in die Umgebung integriert sein. Ein solches System wird als ubiquitäres Rechnersystem bezeichnet und wurde 1991 von Mark Weiser originär vorgeschlagen [Wei91].

Bei der Entwicklung ubiquitärer Rechnersysteme wurde in der Vergangenheit stark auf sogenannte *intelligente Umgebungen*, d.h. elektronisch erweiterte Umgebungen, fokussiert (siehe z.B. [RC00], [PJKF03], [GSSS02]). Hierbei wird in einem eingegrenzten räumlichen Gebiet, beispielsweise einem Zimmer [JFW02] oder Haus [KOA+99], eine geeignete Infrastruktur installiert, die es den im Gebiet vorhandenen Geräten erlaubt, miteinander zu kooperieren. Betritt ein Benutzer die intelligente Umgebung, werden er und seine Geräte automatisch in die Umgebung integriert und können deren Dienste nutzen. Verlässt er das Gebiet, werden seine Geräte aus der Umgebung entfernt. Dieses Architekturmodell ubiquitärer Rechnersysteme erfordert es, vor dem Betrieb des Systems eine geeignete Systeminfrastruktur in Form von Hardware und Software im antizipierten Betriebsumfeld zu installieren. Dies ist mit hohen Investitionskosten verbunden und schränkt die Verwendbarkeit ubiquitärer Systeme auf vorgegebene räumliche Gebiete ein.

In dieser Arbeit wird ein alternatives Architekturmodell für ubiquitäre Rechnersysteme entwickelt, das keine externe Infrastruktur benötigt und auf dem Konzept der sogenannten spontanen Funktionsverbünde beruht. Nach einer Analyse der Herausforderungen, die bei der Entwicklung spontaner Funktionsverbünde auftreten, werden im weiteren Verlauf der Arbeit Verfahren, Konzepte und Algorithmen vorgestellt mit denen diese Herausforderungen überwunden werden können und eine effiziente und flexible Koordination der Geräte eines spontanen Funktionsverbundes ermöglicht wird. Hierzu wird zum einen ein Konzept entwickelt, mit dem eine Flexibilisierung der Gerätekommunikation ermöglicht wird, indem das Kommunikationsmodell einer Anwendung vom Synchronisationsmodell der verwendeten Interoperabilitätsprotokolle entkoppelt wird. Zudem wird eine strategiebasierte dynamische Auswahl der verwendeten Kommunikationsprotokolle vorgeschlagen, mittels derer eine fortlaufende Anpassung des Kommunikationssystems an Wechsel in der Ausführungsumgebung ermöglicht wird. Diese Konzepte werden in eine mikrokernbasierte Verteilungsinfrastruktur integriert, die für ressourcenbeschränkte Geräte entwickelt wurde. Im zweiten Teil der Arbeit werden Verfahren zur dynamischen Erkennung der aktuellen Anwendungsumgebung untersucht und ein Verfahren zur energieeffizienten Erkennung entfernter Dienste und Geräte auf Basis eines Gruppierungsalgorithmus vorgeschlagen.

2 Spontane Funktionsverbünde

In Kapitel 2 der Arbeit wird das Modell des spontanen Funktionsverbunds vorgestellt und näher erläutert. Dieses Modell erlaubt die Erstellung hochflexibler ubiquitärer Rechnersysteme, die ohne die vorherige kostenintensive Installation einer geeigneten Kooperationsinfrastruktur jederzeit und an jedem Ort lauffähig sind. Resultierend hieraus werden die wichtigsten Herausforderungen an ein System zur Unterstützung spontaner Funktionsverbünde diskutiert und Anforderungen an das zu entwickelnde System abgeleitet. Abschließend wird das in dieser Arbeit verwandte Systemmodell beschrieben.
Die grundlegende Idee spontaner Funktionsverbünde ist es, die in der Umgebung eines Nutzers vorliegenden elektronischen Geräte selbstständig so miteinander zu vernetzen, dass diese es dem Nutzer erlauben, verteilte Anwendungen auf ihnen auszuführen. Hierzu bieten die Geräte anderen Geräten in ihrer Umgebung ihre eigenen Ressourcen und Funktionen für die entfernte Nutzung an. Den resultierenden Geräteverbund bezeichnen wir als spontanen Funktionsverbund. Die Bildung bzw. Verwaltung eines spontanen Funktionsverbundes erfolgt hierbei selbstständig durch die an dem Verbund beteiligten Geräte, ohne dass ein Anwender manuelle Einstellungen tätigen muss oder eine externe Infrastruktur verwendet wird. Beispielsweise könnte eine Präsentationsanwendung einen in einem Vorlesungssaal vorhandenen Projektor nutzen um ihre Daten auszugeben, während die Präsentationskontrolle von einem PDA angeboten wird. Zusätzlich könnte eine Anwendung für das Gebäudemanagement selbstständig in die Umgebung eingebettete Aktoren erkennen und diese dazu verwenden, während einer Präsentation den Saal zu verdunkeln.
Aus den beschriebenen Eigenschaften eines Funktionsverbundes können eine Reihe von Her-

ausforderungen abgeleitet werden, die bei der Entwicklung spontaner Funktionsverbünde auftreten. Diese sind insbesondere das hohe Maß an Heterogenität sowohl der beteiligten Geräte und Rechnernetze, als auch der resultierenden Ausführungsumgebungen, die je nach Präsenz verschiedener Geräte sehr unterschiedlich sein können. Eine zweite wichtige Herausforderung bildet die Dynamik des Systems. Die einer Anwendung zu ihrer Ausführung zur Verfügung stehenden Geräte und Funktionen können sich im Zeitverlauf aufgrund der Mobilität der Anwender ständig und unvorhergesehen ändern. Anwendungen müssen sich an veränderte Umgebungen dynamisch anpassen können. Allerdings kann nicht von einer homogenen Verteilung der Systemdynamik ausgegangen werden. Hochdynamische Bereiche mit vielen mobilen Geräten existieren ebenso wie relativ statische Bereiche mit geringer absoluter oder relativer Mobilität. Die dritte Hauptherausforderung für spontane Funktionsverbünde bildet die starke Ressourcenbeschränkung vieler der beteiligten Geräte aufgrund von Größen-, Gewichts- und Kostenrestriktionen. Insbesondere gilt dies für batteriebetriebene Geräte, die nur einen eingeschränkten Energievorrat zur Verfügung haben.
Um die diskutierten Herausforderungen zu überwinden, müssen spontane Funktionsverbünde eine Reihe von Anforderungen erfüllen. Zum einen wird ein Verfahren benötigt, mit dem die aktuell vorliegende Anwendungsumgebung bezüglich der im Verbund verfügbaren Geräte und der von diesen bereitgestellten Funktionen fortlaufend erfasst und jede Veränderung identifiziert werden kann. Zudem muss eine interoperable und anpassbare Kommunikationsunterstützung für Geräte eines spontanen Verbunds bereitgestellt werden, die die Kooperation zwischen heterogenen Knoten ermöglicht und sich bei Änderungen in der Anwendungsumgebung an diese anpasst, so dass eine Kommunikation in möglichst vielen Szenarien möglich ist. Hierzu kann z.B. bei Bedarf zwischen verschiedenen Kommunikationstechnologien umgeschaltet werden. Da spontane Funktionsverbünde ohne unterstützende externe Infrastrukturkomponenten auskommen sollen, müssen sowohl die Erkennung als auch die Nutzung entfernter Geräte dezentral erfolgen. Zudem muss das System an die verschiedenen zum Einsatz kommenden Geräte anpassbar sein um auf ressourcenschwachen Geräten wie beispielsweise Sensorknoten ebenso lauffähig zu sein wie auf Rechnern mit vielen Ressourcen. Insbesondere muss der Betrieb des Funktionsverbundes auf den resultierenden Energieverbrauch optimiert sein, um die Lebenszeit batteriebetriebener Geräte zu erhöhen.
Das in dieser Arbeit verwandte Systemmodell beschreibt ein ubiquitäres Rechnersystem als ein System bestehend aus elektronischen Geräten, aus den zur Vernetzung von Geräten verwendeten Rechnernetzen, aus von Geräten in Form von Diensten angebotenen Funktionen und aus die Geräte nutzenden Anwendern. Geräte (bzw. Knoten) sind mobil und enthalten jeweils eine oder mehrere drahtlose Kommunikationsschnittstellen, eine Batterie und einen knoteninternen Zeitgeber. Aufgrund der großen Anzahl von Knoten, sowie deren Integration in Alltagsgegenstände können diese nicht von Anwendern vor ihrer Verwendung manuell gestartet werden, sondern werden als dauerhaft in Betrieb angenommen, bis ihre Batterien leer sind. Knoten bieten zwei Betriebsmodi: *schlafend* und *aktiv*. Im schlafenden Zustand verbaucht ein Knoten sehr

viel weniger Energie als im aktiven Zustand, ist aber nicht in der Lage Berechnungen durchzuführen oder zu kommunizieren. Der Knoten kann programmatisch jederzeit in den Zustand schlafend versetzt werden. Die Reaktivierung des Knotens erfolgt nach einer vorgegebenen Zeitspanne durch den knoteninternen Zeitgeber. Weitere Möglichkeiten zur Reaktivierung werden nicht vorausgesetzt. Jeder Knoten kann über eine vorab festgelegte systemweit eindeutige Identifikationsnummer eindeutig bestimmt werden. Zur Vernetzung der Knoten werden drahtlose Kommunikationstechnologien verwendet, mit deren Hilfe die Knoten sogenannte mobile ad-hoc Netze bilden. Die Topologie der Rechnernetze kann sich insbesondere bedingt durch die Knotenmobilität jederzeit ändern. Die verfügbare Dienstgüte, z.B. Datenrate und Verzögerung, einer Kommunikation kann ebenfalls jederzeit wechseln. Da Knoten teilweise mit mehreren Kommunikationsschnittstellen versehen sind, können sich mehrere Netze überlappen. Anwender sind mobil, führen ein oder mehrere Geräte mit sich und greifen auf das System zu, um Anwendungen ausführen zu lassen. Im Rahmen dieser Arbeit wird ein kooperatives Anwendermodell angenommen, d.h. Anwender sind bereit, die Dienste ihrer eigenen Geräte anderen Anwendern zur Verfügung zu stellen.

3 Eine Verteilungsinfrastruktur für spontane Funktionsverbünde

Die Entwicklung eines auf spontanen Funktionsverbünden basierenden ubiquitären Rechnersystems, das alle diskutierten Anforderungen erfüllt, ist eine komplexe und fehlerträchtige Aufgabe. Daher wird in Kapitel 3 der Arbeit eine Verteilungsinfrastruktur namens *BASE* entworfen und diskutiert, die als Basis einer solchen Entwicklung dienen kann. BASE bietet einen konzeptionellen und architektonischen Rahmen, in den verschiedene Algorithmen und Verfahren zur automatisierten und flexiblen Kooperation von Geräten eines spontanen Funktionsverbundes integriert werden können, beispielsweise verschiedene Strategien zur Anpassung der verwendeten Kommunikationsprotokolle und verschiedene Diensterkennungsverfahren.
Bei der Entwicklung von BASE wurden eine Reihe von Entwurfsentscheidungen gefällt, die im Folgenden kurz erläutert werden. Die Programmierschnittstelle von BASE basiert auf dem Konzept von Diensten. Jedes Gerät bietet seine lokalen Funktionalitäten anderen Geräten in Form von Diensten an. Der Zugriff auf Dienste erfolgt unabhängig von deren Lokation, was es ermöglicht, bei Bedarf zwischen lokalen und entfernten Diensten zu wechseln und sich damit an welchselnde Umgebungen anzupassen. Davon unabhängig können Entwickler die Lokation eines Dienstes abfragen, z.B. um explizit einen lokal ausgeführten Dienst auszuwählen. Eine weitere Entwurfsentscheidung in BASE ist die Entkopplung des von einer Anwendung verwendeten Kommunikationsmodells von den verwendeten Interoperabilitätsprotokollen. Hierdurch kann flexibler zwischen verschiedenen Interoperabilitätsprotokollen gewechselt werden. Verfügt ein Gerät über mehrere Kommunikationsschnittstellen bzw. -protokolle, muss zur Laufzeit ein geeigneter Protokollstapel in Abhängigkeit von der gegebenen Umgebungssituation ausgewählt werden. Diese Auswahl erfolgt durch BASE, wodurch die Anwendung von den

spezifisch vorhandenen Protokollen unabhängig bleibt. Der Anwendungsentwickler kann die Auswahl durch die Spezifikation anwendungsabhängiger Anforderungen an die Kommunikation beeinflussen. Die eigentliche Auswahl erfolgt durch eine austauschbare Selektionsstrategie. Dies ermöglicht es, abhängig von den auf einem Gerät verfügbaren Ressourcen, unterschiedlich komplexe Strategien zu verwenden und somit z.B. auf einem ressourcenschwachen Gerät eine Strategie auf Basis einer statisch festgelegten Reihenfolge zwischen Protokollen einzusetzen, während auf einem ressourcenstarken Gerät eine komplexere Strategie Verwendung findet, die z.B. die derzeitige Signalstärke in die Selektion einbezieht. Die Middleware ist in Form eines erweiterbaren minimalen Kerns aufgebaut, der an Geräte und Umgebungen angepasst werden kann. Dadurch kann BASE auf verschiedenartigsten Geräten eingesetzt werden und deren Ressourcen effektiv nutzen ohne dass eine große Zahl spezieller Middlewarevarianten gepflegt werden muss. Um den hohen Ansprüchen bezüglich der Gerätelaufzeit in ubiquitären Rechnersystemen zu genügen, muss BASE auf einen energieeffizienten Betrieb der Geräte optimiert sein. Der hierzu in BASE gewählte Ansatz beruht auf zwei Beobachtungen. Erstens warten Geräte in ubiquitären Systemen oft aktiv darauf, genutzt zu werden, da sie aufgrund ihrer Vielzahl und Integration in Alltagsgegenstände nicht manuell ein- und ausgeschaltet werden. Zweitens haben aktiv wartende Geräte einen erheblichen Energieverbrauch [SBS02]. Um diesen Verbrauch zu reduzieren, erlaubt es BASE, nichtverwendete Geräte in einen energiesparenden Modus zu versetzen. Zusätzliche Einsparungen sind durch die Integration energiesparender Kommunikationsprotokolle in BASE und deren Selektion zur Laufzeit erreichbar.

BASE kann in drei Teilbereiche eingeteilt werden: den Anwendungsbereich, den Mikro-Broker-Bereich und den Erweiterungsbereich. Der Anwendungsbereich besteht aus Klienten, Anwendungsdiensten und Systemdiensten, z.B. dem Dienstverzeichnis. Das Dienstverzeichnis verwaltet Informationen über alle lokal verfügbaren Dienste eines Gerätes und bietet die Möglichkeit, entfernte Dienste zu suchen. Eine Dienstsuche kann entweder über den Namen des gewünschten Dienstes erfolgen oder über die Angabe gewünschter funktionaler und nichtfunktionaler Diensteigenschaften. BASE bietet zwei Programmierschnittstellen. Zum einen können Stellvertreterobjekte verwendet werden, um analog zu existierenden Verteilungsinfrastrukturen entfernte Dienste mittels Stubs und Skeletons zu verwenden. Zum anderen können mittels sogenannter Invocations Aufrufe an Dienste zur Laufzeit vom Entwickler manuell zusammengestellt und versandt werden, was deutlich flexibler aber auch komplexer als die erste Möglichkeit ist. Intern bilden Stellvertreterobjekte Aufrufe wiederum auf Invocations ab. Zentraler Bestandteil der Verteilungsinfrastruktur ist der Mikro-Broker, der aus dem Invocation-Broker, dem Plugin-Manager, dem Geräteverzeichnis (device registry) und dem Objektverzeichnis (object registry) besteht. Der Invocation-Broker nimmt Invocations entgegen, leitet diese an einen lokalen Dienst bzw. ein geeignetes Kommunikationsprotokoll weiter und realisiert gegenüber der Anwendung das von dieser gewünschte Synchronisationsmodell. Der Plugin-Manager implementiert die Erweiterbarkeit der Middleware durch sogenannte Plugins. Ein Plugin kann beispielsweise ein spezielles Kommunikationsprotokoll beinhalten. Das Objektverzeichnis ent-

hält alle Informationen, die zur Vermittlung eingehender Aufrufe notwendig ist, z.B. an wen ein Aufruf weiterzuleiten ist. Das Geräteverzeichnis beinhaltet Informationen über alle anderen bekannten Geräte im spontanen Funktionsverbund, z.B. über welche Protokolle mit diesen kommuniziert werden kann. Der Erweiterungsbereich von BASE besteht aus den derzeit am Plugin-Manager angemeldeten Plugins und einer austauschbaren Selektionsstrategie, die zur Auswahl geeigneter Plugins zur Laufzeit herangezogen wird. Die Plugin-Auswahl kann vom Anwendungsentwickler durch die Spezifikation verschiedener gewünschter Eigenschaften der selektierten Plugins in Form einer Menge von Name-Wert-Paaren gesteuert werden. Zudem bieten Plugins Zugriff auf ihren derzeitigen Zustand, z.B. die derzeitige Signalstärke bei einem Kommunikationsprotokoll, um es einer Strategie zu erlauben, auch solche Informationen für Plugin-übergreifende Optimierungen zu verwenden. Bezüglich der Granularität von Plugins bietet BASE verschiedene Möglichkeiten. Zum einen kann ein Plugin einen kompletten Protokollstapel enthalten, was eine effiziente, integrierte Implementierung erlaubt. Zum anderen können Plugins nur Teile eines Stapels implementieren, wodurch ein flexibler anpassbares System entsteht. In diesem Fall muss die Selektionsstrategie eine Kette von Plugins auswählen und zu einem Stapel zusammenfügen. Um die Entwicklung und Auswahl solcher Plugins zu vereinfachen, spezifiziert BASE eine eigene Schichtenarchitektur für Plugins, die aus Plugins auf semantischer Ebene, Serialisierungs-Plugins, Modifikations-Plugins und Übertragungs-Plugins besteht. Semantische Plugins realisieren verschiedene Aufrufsemantiken, z.B. dass ein Aufruf maximal einmal durchgeführt wird. Hierzu erzeugen und versenden sie ggf. mehrere Invocations. Serialisierungs-Plugins bilden einzelne Invocations auf Byteketten ab und umgekehrt. Modifikations-Plugins verändern diese Byteketten, z.B. durch Kompression oder Verschlüsselung. Übertragungs-Plugins versenden und empfangen Invocations. Plugins können hierbei die Funktionalität einer oder mehrerer dieser Schichten realisieren.
Abschließend werden in diesem Kapitel existierende Arbeiten im Bereich der Verteilungsinfrastrukturen untersucht und die vorliegende Arbeit gegen diese abgegrenzt. Hierbei werden zunächst konventionelle Verteilungsinfrastrukturen (z.B. [Obj04], [Sun02], [EE98]) behandelt, die zu ressourcenintensiv und unflexibel für die Verwendung in spontanen Funktionsverbünden sind. Eine höhere Flexibilität bezüglich des Verhaltens der Verteilungsinfrastruktur bieten Reflexive Infrastrukturen (z.B. [Led99], [RKC99], [BG00], [BCRP00], [BCA$^+$01], [RKC01]). Diese Infrastrukturen sind in der Lage, ihr Verhalten zur Laufzeit an verschiedene Anwendungsanforderungen anzupassen. Allerdings konzentrieren sich diese Infrastukturen meist auf leistungsfähige Anpassungsschnittstellen und sind nur eingeschränkt für ressourcenarme Geräte einsetzbar. Solche Geräte werden von Middlewaresystemen für ressourcenarme Geräte addressiert, wobei Verteilungsinfrastrukturen entweder nur eine funktionale Teilmenge anbieten können (z.B. [Obj02], [RSC$^+$99], [Sch04]), oder die Infrastruktur in einzelne Komponenten aufgeteilt werden kann, die bei Bedarf hinzugefügt oder entfernt werden können (z.B. [RKC01]). Die in dieser Arbeit entwickelte Verteilungsinfrastruktur BASE folgt dem Konzept der Aufteilung in mehrere Komponenten, wodurch eine sehr flexible Anpassbarkeit an

verschiedene Geräte möglich ist.

4 Diensterkennung in spontanen Funktionsverbünden

Bevor ein Dienst eines spontanen Funktionsverbunds genutzt werden kann, muss dieser zunächst gefunden werden. Hierfür wird ein Diensterkennungsverfahren verwendet. In den Kapiteln 4–7 wird ein solches Verfahren namens SANDMAN vorgestellt, das speziell für die Verwendung in spontanen Funktionsverbünden entwickelt wurde. Hierbei bildet Kapitel 4 den konzeptionellen Rahmen des Verfahrens, während sich die folgenden Kapitel jeweils einem der drei Teilsysteme von SANDMAN widmen. Zunächst werden die Ziele des zu entwickelnden Erkennungsverfahrens beschrieben, das eine schnelle, korrekte und vollständige Erkennung vorhandener Dienste anstrebt. Danach werden verschiedene Entwurfsentscheidungen des Verfahrens diskutiert, insbesondere die gewählte Systemarchitektur und das Informationsverteilungsmuster. SANDMAN basiert auf einer hierarchischen Systemarchitektur bei der Vermittlerknoten die Diensterkennung für Gruppen von Geräten übernehmen. Dies erlaubt es den anderen Knoten einer Gruppe, sich zeitweise zu deaktivieren und so Energie zu sparen. Allerdings müssen die Vermittler kontinuierlich dynamisch gewählt werden, um das System an Änderungen in der Netzwerktopologie anzupassen. Dies verursacht zusätzlichen Audwand, weshalb eine möglichst stabile Einteilung in Gruppen und Zuordnung zu Vermittlern anzustreben ist. Bei der Informationsverteilung wird ein reaktives Verteilungsmuster gewählt, da hierdurch die suchenden Knoten mittels der Suchhäufigkeit den entstehenden Aufwand steuern können. Das resultierende Diensterkennungsverfahren ist in drei Teilsysteme eingeteilt, die miteinander interagieren. Dies reduziert die Komplexität jedes einzelnen Teilsystems.
Die Einteilung der Geräte in Gruppen und die Zuordung einer Gruppe zu einem Vermittler ist Aufgabe des Gruppenmanagements. SANDMAN ist unabhängig von dem spezifischen gewählten Gruppenmanagementverfahren. Im Rahmen der Arbeit wird in Kapitel 5 ein für spontane Funktionsverbünde geeignetes Gruppenmanagementverfahren entwickelt. Ziel des Verfahrens ist es, möglichst stabile Gruppen zu wählen, in denen alle Knoten ausser dem Vermittler deaktiviert werden können, ohne die Konnektivität des zugrundeliegenden Kommunikationsnetzwerks zu beeinträchtigen. Hierzu wird die Nachbarschaft eines Knotens periodisch erfasst und zwei Knoten werden gruppiert, wenn sie sich über einen gegebenen Zeitraum hinaus in direkter Nachbarschaft befinden. Zudem müssen beide Knoten die selbe Nachbarschaft aufweisen. Dies führt dazu, dass Knoten nur dann gruppiert werden, wenn sie sich über einen längeren Zeitraum sehr nah beieinander bewegen. Der Vermittler einer Gruppe wird aus den Teilnehmern der Gruppe gewählt, indem der Knoten mit der längsten verbleibenden Lebensdauer gewählt wird, wodurch energiearme Geräte entlastet werden. Um das Verfahren fair zu machen und einzelne Knoten nicht übermäßig zu belasten, wird die Vermittlerrolle gewechselt, falls die Lebensdauer des Vermittlers deutlich unter die eines anderen Gruppenmitglieds sinkt.
Aufbauend auf der Gruppierung der Knoten, definiert das Dienstmanagement in Kapitel 6, wie

Dienstanbieter und -nutzer mit einem Vermittler interagieren. Insbesondere wird festgelegt, wie ein Dienst beim Vermittler registriert und deregistriert wird, wie seine Beschreibung geändert werden kann und wie ein Dienstnutzer eine Suchanfrage beim Vermittler initiiert. Suchanfragen können mit einer Suchreichweite parametrisiert werden, so dass eine Suche auf dem lokalen Knoten, in der eigenen Gruppe oder in der gesamten Umgebung erfolgen kann. Hierdurch kann der entstehende Energieverbrauch für die Suche gesteuert werden.
Schließlich wird in Kapitel 7 das in SANDMAN integrierte Energiemanagement erläutert. Dieses ist dafür zuständig zu entscheiden, wann und für wie lange ein gegebener Knoten in seinen Energiesparmodus geschaltet wird. Dies muss vorab berechnet werden, da ein Knoten, sobald er in den Energiesparmodus gewechselt ist, nicht mehr vorzeitig reaktiviert werden kann. Es werden zwei Ansätze für das Energiemanagement vorgestellt. Der erste Ansatz berechnet Deaktivierungszeitpunkt und -länge basierend auf lokal auf einem einzelnen Knoten vorliegenden Informationen, beispielsweise wann dieser zuletzt verwendet wurde. Der zweite Ansatz zieht zusätzlich den Zustand aller anderen Knoten einer Gruppe für die Berechnung hinzu. Dies erlaubt es, Knoten, die alternativ nutzbare Dienste anbieten, versetzt zu deaktivieren und somit bei gleicher Deaktivierungslänge deutlich kürzere Nutzungsverzögerungen zu realisieren.
Zusätzlich zu der in Kapitel 3 vorgenommenen Diskussion verwandter Arbeiten, werden in dieser Dissertation auch existierende Diensterkennungsverfahren untersucht. Diensterkennungsverfahren können in Vermittlerbasierte Verfahren (z.B. [Sun01], [AWSBL99]) und Peerbasierte Verfahren (z.B. [MNTW01], [Nid01]) klassifiziert werden. Bei Vermittlerbasierten Verfahren werden einzelne Geräte mit der Aufgabe betraut, ein Dienstverzeichnis zu führen, in das sich Dienstanbieter eintragen können und bei dem Dienstnutzer ihre Suchanfragen stellen können. Allerdings sind diese Verfahren abhängig von der Erreichbarkeit dieser Vermittler und daher in spontanen Funktionsverbünden nicht einsetzbar. Peerbasierte Verfahren benötigen keine Vermittler. Eine Dienstsuche erfolgt, indem das suchende Gerät allen potentiellen Anbietern seine Suchanfrage übermittelt. Alternativ kann ein Anbieter allen potentiellen Nutzern sein Angebot mitteilen. Dies kann in spontanen Funktionsverbünden zu einem hohen Datenaufkommen führen. Zudem müssen Geräte bei den meisten Peerbasierten Verfahren immer aktiv sein, um Anfragen bzw. Angebote zu empfangen, was zusätzliche Energieressourcen benötigt. Existierende Diensterkennungsverfahren für ressourcenschwache Geräte (z.B. [HLMW02], [Sun06], [DS01]) konzentrieren sich auf Ressourcen wir CPU oder Speicher und bieten ebenfalls keine Unterstützung für einen energieeffizienten Betrieb. Eine Ausnahme hiervon bildet das Diensterkennungsverfahren von DEAPspace [Nid01] [Nid00]. Dieses peerbasierte Verfahren erlaubt es, Geräte die sich miteinander synchronisiert haben zeitweise zu deaktivieren, wobei Dienstsuchen mit einem auf allen Geräten vollständig replizierten Dienstverzeichnis durchgeführt werden. Allerdings ist DEAPspace nur für Geräte in direkter Kommunikationsreichweite einsetzbar und erzielt Deaktivierungsanteile von maximal 50 % [Nid00]. Der in dieser Arbeit vorgestellte Ansatz kann auch in größeren spontanen Funktionsverbünden eingesetzt werden und erreicht deutlich höhere Deaktivierungszeiten.

Auch im Bereich der Gruppenmanagementverfahren gibt es verwandte Arbeiten. Es werden verschiedene Verfahren für die energieeffiziente Kommunikation auf Basis von Gerätedeaktivierung und Cluster-Bildung (z.B. [IEE99], [YHE02], [XHE01], [XBM$^+$03], [CJBM02]) und Verfahren für die Bildung der Cluster selbst untersucht. Erstere arbeiten, anders als das in dieser Arbeit vorgestellte Verfahren, auf tieferen Schichten des Protokollstapels, z.B. auf Routingebene. Dadurch sind sie nicht in der Lage, abhängig von der Art der empfangenen Nachricht zu agieren, z.B. indem sie Suchanfragen direkt beantworten anstatt diese zwischenzuspeichern. Die untersuchten Clustering-Verfahren können entsprechend der von ihnen primär addressierten Anforderungen klassifiziert werden. Verfahren für die Erzeugung stabiler Cluster (z.B. [CC-CG97], [BKL01]) sind nicht in der Lage zeitweise deaktivierte Geräte zu verarbeiten. Energieeffiziente Verfahren (z.B. [XHE01], [CJBM02], [XBM$^+$03]) bilden keine stabilen Cluster und benötigen synchronisierte Aufwachzeiten der Geräte.

5 Evaluation

Kapitel 8 der Arbeit beschreibt die Evaluation der vorgeschlagenen Konzepte und Algorithmen. Hierbei werden zunächst die zur Evaluation verwendeten Leistungsmetriken erläutert. Neben den erzielbaren Energieeinsparungen sind dies die Erkennungsverzögerung, die angibt, wie schnell ein neuer Dienst im System erkannt wird, die Erkennungspräzision, die den Anteil korrekt erkannter Dienste beschreibt, die Erkennungvollständigkeit, die als Maß dafür verwendet werden kann, welcher Anteil verfügbarer Dienste erkannt wurde, sowie der für den Betrieb des Systems notwendige Nachrichtenaufwand. Anschließend werden die vorhandenen Systemparameter beschrieben, wobei diese in Parameter des Mobilitätsmodells, des Energiemodells, des Umgebungsmodells und des Verfahrens SANDMAN eingeteilt werden und Abhängigkeiten zwischen Parametern diskutiert werden. Darauf aufbauend wird eine Reihe analytischer Modelle entworfen, mit deren Hilfe Abhängigkeiten zwischen verschiedenen Parametern genauer charakterisiert werden können. Hierbei wird jeweils auf eine der zuvor diskutierten Leistungsmetriken fokussiert und für diese geeignete Modelle entwickelt. Schließlich wird ergänzend zu der analytischen Betrachtung der Systemleistung eine experimentelle Evaluation präsentiert. Hierbei wurde auf Basis der Emulationsumgebung NET eine reale Implementierung der in dieser Arbeit vorgeschlagenen Verfahren erstellt und in verschiedenen Szenarien ausgeführt. Zu diesem Zweck wurde NET um Mechanismen zur Deaktivierung von Geräten sowie zur Messung des Energieverbrauchs eines Gerätes erweitert. Zudem wurde ein geeignetes Mobilitätsmodell ausgewählt, das es erlaubt, Szenarien zu realisieren, in denen mehrere Geräte sich gemeinsam bewegen, das sogenannte RPGM. Analog zu dem bei der analytischen Evaluation gewählten Vorgehen wird auch bei der experimentellen Evaluation auf jeweils eine Leistungsmetrik fokussiert und die erzielten Ergebnisse diskutiert.

6 Zusammenfassung und Ausblick

Die Arbeit schließt in Kapitel 10 mit einer kurzen Zusammenfassung und einem Ausblick auf weitere sich aus dieser Arbeit ergebende Forschungsvorhaben. Hierbei wird unter anderem auf die Möglichkeit eingegangen, weitere Verfahren für das Gruppenmanagement sowie die Planung der Deaktivierung von Geräten in SANDMAN zu integrieren. Zudem kann der Vermittler für weitere Funktionen eines spontanen Funktionsverbundes verwendet werden, z.B. für die Dienstnutzung. Schließlich ist es denkbar, das in dieser Arbeit entwickelte Modell des spontanen Funktionsverbundes zu erweitern und mit dem Modell intelligenter Umgebungen zu einem hybriden Systemmodell zu integrieren.

1 Introduction

1.1 Motivation

In the last years, mobile computing devices have begun to pervade our daily lives. Millions of people are using mobile phones, personal digital assistants (PDAs), personal navigation assistants (PNAs), or laptops. Driven by more and more powerful devices and continuously smaller form factors, computing power becomes readily available everywhere and anytime. In addition, different wireless networking technologies to connect these mobile devices have emerged. Wide area networking technologies like UMTS [Soi00] can be used by travelers to check their email while on the move. Local and personal area networking technologies like IEEE 802.11 [IEE99] or Bluetooth [Blu04] allow to easily connect mobile phones with their headsets or mobile game consoles with each other [Nin06].

Ubiquitous or Pervasive Computing [Wei91] greatly enhances such scenarios. In a Pervasive Computing system, a multitude of small embedded and typically highly specialized computing devices are integrated into our everyday environment and allow ubiquitous and unobtrusive access to computing power. Often, users may not even be aware of the fact, that they are using a computer system anymore [Wei93]. Examples of such devices are an electronically enhanced pen [Slo06] or smart clothing [Man97]. To execute applications, the devices are networked and cooperate with each other, e.g., by forming so called wireless Mobile Ad hoc Networks (MANETs).

In the past, Pervasive Computing has often been addressed by developing so-called Smart Environments, e.g., Gaia [RC00], iRos [PJKF03], Aura [GSSS02], and others (e.g. [ACH+01], [CPW+99], [Moz98], [RC00]), as shown in Figure 1.1. Smart Environments provide a fixed infrastructure to manage a predefined spatial area, such as a meeting room [JFW02] or a house [KOA+99]. The infrastructure mostly consists of a number of stationary, networked devices, e.g., wall-mounted displays and storage servers. Basic system services, like device coordination or discovery, are provided by the Smart Environment's infrastructure. On top of this stable infrastructure, users and their mobile devices are dynamically integrated into the system. If a user enters the Smart Environment, his devices are integrated into it and can permanently access its services using some wireless communication technology, e.g., IEEE 802.11. Once he leaves it, his devices are removed from the Smart Environment, too. To summarize, Smart Environments concentrate on specific spatial areas and rely on a preinstalled computing infrastructure.

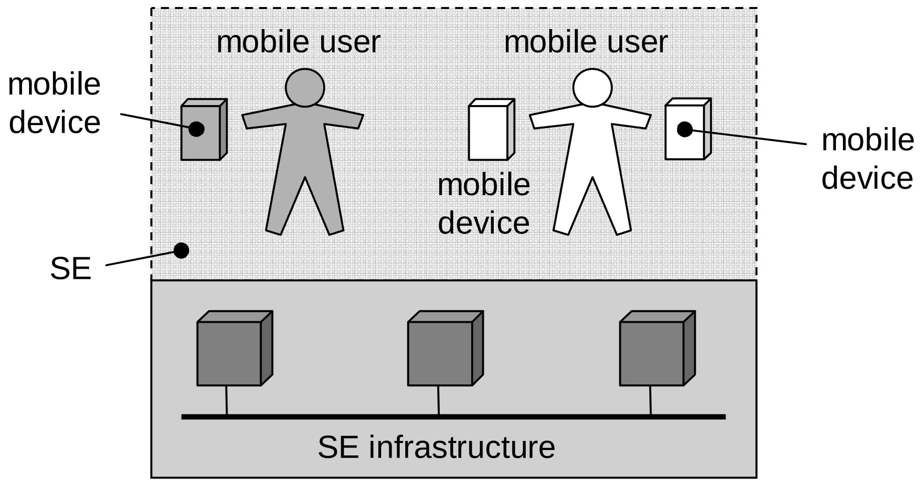


Figure 1.1: Smart Environments

As an example, the Gaia system [RC00] allows to enhance rooms with computers to ActiveSpaces. Gaia provides an infrastructure to spontaneously connect devices offering or using services registered in Gaia. To integrate existing systems, like CORBA, interaction between application objects is done via the Unified Object Bus [RC01], which is layered on top of these systems. As essential system services, such as discovery and lookup, are provided by the Gaia infrastructure, mobile devices cannot cooperate autonomously without the infrastructure.

The main drawback of a Smart Environment is its dependency of the preinstalled infrastructure. If no such infrastructure is accessible at a given location, no device cooperation is possible, and the devices have to fall back to an isolated operation mode. This could be the case, because no infrastructure has been installed or because the infrastructure is not trustworthy, like in a meeting room of a competing company. In addition, the installation and maintenance of a Smart Environment infrastructure may require large monetary investments. Any service provider willing to place such an investment must be sure that a suitable return on investment will be achieved. Small companies or private households will not be able to invest such large sums and will wait for cheaper solutions, delaying the distribution of Pervasive Computing systems.

In this dissertation, we propose a different peer-to-peer-based model for Pervasive Computing systems, that complements Smart Environments by offering means for devices to cooperate without the need for any external infrastructure. We call this model *Smart Peer Groups*. The main idea of Smart Peer Groups is to shift the main viewpoint from an electronically enhanced predefined spatial area to the mobile users and their vicinity. This leads to a kind of electronic aura surrounding users that consists of devices carried by the users, e.g., sensors or PDAs, as well as nodes in the user's vicinity, e.g., a wall mounted display. A Smart Peer Group relies solely on the spontaneous cooperation of the interconnected peer devices without support from any external infrastructure. Any system service must therefore be provided by the mobile devices themselves, leading to a peer-based coordination model. Devices can directly access and use each others services as long as they are willing to cooperate and stay in communication range. For devices carried by different users this may lead to frequent connectivity changes. Although multi-hop routing can be used to access far away devices, users are typically more interested in information and services in their direct physical proximity, e.g., for in-/output services or due to communication delay and energy consumption. To summarize, a Smart Peer Group consists of a dynamic set of devices that cooperate spontaneously and without any preinstalled external infrastructure.

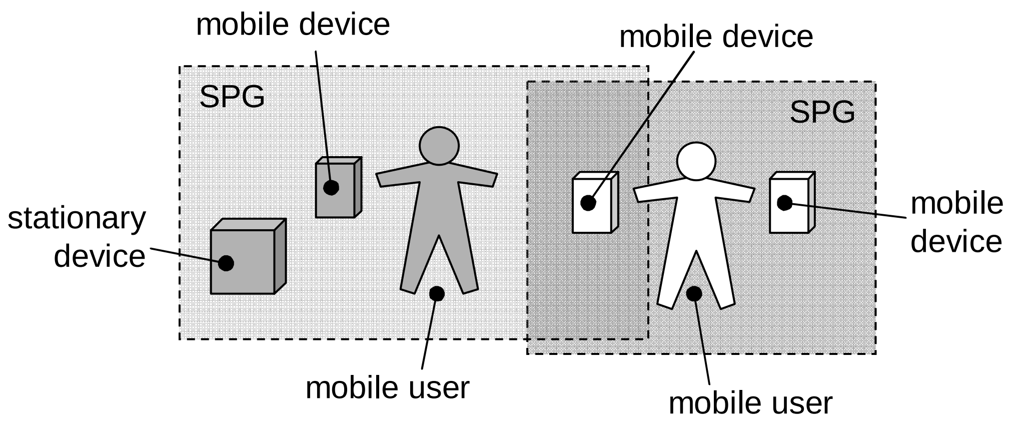


Figure 1.2: Smart Peer Groups

This approach offers a number of potential advantages. First, Smart Peer Groups allow cooperation between devices at any place and any time, even in areas, where no external infrastructure support is accessible or access is forbidden, e.g., due to security reasons. Second, Smart Peer Groups provide a good possibility to gradually build up a large Pervasive Computing system as they can be readily used without prior investment in infrastructure deployment. Third, in areas in which an infrastructure, such as a Smart Environment, exists, the infrastructure may be included into a Smart Peer Group by modeling it as a very powerful device. This allows the development of simple hybrid systems, which integrate Smart Environments and Smart Peer Groups.

However, the development of a Smart Peer Group system is complex, due to its highly dynamic and in general unpredictable nature. As devices may enter or leave the Smart Peer Group at any time, and often will do so, the system must constantly adapt itself and all its applications to the resulting changes in the environment. Additionally, the system cannot assume the presence of any centralized system services or dedicated high performance server, as Smart Environments do. Every device should be able to work on its own or in combination with an arbitrary set of other devices. This should be the case for high performance computers like, e.g., laptops as well as for resource-poor devices like sensor nodes. Therefore, a Smart Peer Group must be able to operate on very scarce resources like compute power, communication bandwidth or energy. In fact, energy becomes a major factor for Smart Peer Group systems that are formed by battery powered mobile devices and must therefore be handled with much care. For a more detailed discussion of the characteristics of Smart Peer Group systems and the resulting challenges refer to Chapter 2.

1.2 Classification

The development of applications for such environments is a non-trivial task. Induced by the mobility, fluctuating network connectivity or changing physical context, the hardware and software resources available to an application at runtime are continuously fluctuating. As a result, applications need to adapt themselves dynamically to their ever-changing execution environment. To enable such adaptive applications, different levels of support are needed. In the following, we describe these levels and discuss them briefly.

Ad hoc Networking

The first step in supporting adaptive applications is to enable devices to communicate with each other. Whenever two or more devices that are equipped with the same wireless communication technology enter their communication range, they are able to send data to each other. Depending on the availability of a suitable routing algorithm, communication can be possible between neighboring devices only or between any two devices in the same network by using multiple forwarding nodes between them. Such multi-hop routing may also enable communication between devices with different network interfaces. A lot of work has been done in the area of ad hoc networking, including wireless communication technologies (e.g., [SCI$^+$01]), MAC protocols (e.g., [SR98] [SCS00] [YHE02]), and routing protocols (e.g., [WSR98] [XHE01] [CJBM02]).

Spontaneous Cooperation

While ad hoc networking enables devices to exchange data with each other, cooperating devices want to access functionality on each other, e.g., to output text on a remote display. To do so, additional support is required. A suitable abstraction for accessing remote functionalities must be provided, as well as protocols to discover them. This is typically done by defining a suitable service model, which allows to model a device's functionalities as a set of services which the device offers via well defined interfaces to other devices. A communication middleware implements this model and provides service discovery mechanisms and interoperability protocols to find and access the services. Spontaneous cooperation is an important area of current research. Different approaches are discussed in Chapter 3.4.

Automated Application Adaptation

In addition to ad hoc networking and spontaneous cooperation, further support for adaptive applications can be offered by providing automated application adaptation. While cooperation mechanisms allow an application to manually react to changes in its execution environment, e.g., by reselecting some used service, adaptation support allows the system to automatically adapt the application. To do so, a suitable application model, which reflects the requirements on the different application parts at runtime as well as a suitable execution environment must be provided. One possible approach is to define a component based application model and a component container to realize this model. In addition, to allow context dependent adaptations a suitable context service could be provided. This area is the focus of a number of current research projects, e.g., [GDL$^+$01] [SPP$^+$03] [BHS04].

Discussion

This thesis reports on research conducted in the peer-to-peer pervasive computing (3PC) project [BHSM04] at the Universität Stuttgart. The 3PC project concentrates on providing system support for spontaneous cooperation and automated application adaptation. Communication support has been studied extensively in the related work. We build upon this work and assume that suitable communication support is available. The scope of this work is restricted to spontaneous cooperation (see Figure 1.3). Higher level support for automated adaptation and context awareness is subject to other work done in the 3PC project (see, e.g., [HSUB05], [UHSR06], [HHSB07]). However, the provided basic support must be suitable to act as basis for such higher level support.

	Smart Environments	Smart Peer Groups
Ad hoc networking		
Spontaneous cooperation		**topic of this dissertation**
Automated application adaptation		

Figure 1.3: Classification

1.3 Contributions

In this dissertation, we present the Smart Peer Group model and analyze the specific characteristics of this system class. In addition, we propose a number of concepts and algorithms to develop Smart Peer Group-based Pervasive Computing systems. More specifically, the main contributions of this dissertation are as follows:

First, we present peer-to-peer-based coordination of ubiquitous computing environments. Requirements that are specific to peer-to-peer-based coordination are derived and discussed. Based on these requirements, concepts and algorithms for the flexible and efficient coordination with respect to communication and service discovery are presented.

Communication in such environments is challenged by the fluctuation of communication technology due to user and device mobility and the absence of a dedicated communications infrastructure. This dissertation introduces a concept for flexible communication support by decoupling the communication model of the application from the synchronization model of the interoperability protocol and shifts the responsibility for synchronization into the middleware. Strategy-based adaptation allows the automatic selection of communication protocols under given constraints, e.g., latency, bandwidth, energy. These concepts are integrated into a micro-kernel-based middleware architecture that allows flexible and dynamic extension of functionality at runtime.

Peer-based systems have to monitor their environment constantly in order to detect new and leaving services. Since many devices in such settings will be battery powered, energy is a limiting factor and thus a crucial resource. Wireless communication is a major factor of a node's power consumption. This dissertation contributes to energy efficient service discovery by introducing a cluster-based coordination scheme that is integrated into the overall architecture in order to prevent adaptation anomalies and allows for cross-layer optimizations. A discovery framework that is built on these concepts is developed. Diverse protocols for cluster management and service scheduling can be integrated. One example for a clustering protocol that identifies stable network parts in order to group devices accordingly and its integration into the framework is presented.

1.4 Structure

The remainder of this dissertation is structured as follows: in Chapter 2, we discuss Smart Peer Groups in more detail. We present their characteristics and derive the main requirements for our work. In addition, we describe the system model used in this dissertation. In Chapter 3 we develop a communication middleware for Smart Peer Groups, that acts as the basis of our further work. Building upon this middleware, Chapter 4 introduces a novel service discovery system for Smart Peer Groups and Chapter 5 adds a suitable cluster management protocol to be used with this service discovery system. More details about finding services and about deactivating devices are given in Chapters 6 and 7. We evaluate our work in Chapter 8. Concluding the dissertation, Chapter 9 summarizes our findings and gives further research directions.

2 Smart Peer Groups

This chapter introduces the concept of Smart Peer Groups. After a short motivation we discuss the main challenges of Smart Peer Group-based systems and derive requirements for our approach. Finally, we introduce the system model used in this work and finish the chapter with a short summary.

2.1 The Smart Peer Group Model

The Smart Peer Group model provides a conceptional framework for the development of highly flexible Pervasive Computing systems. In contrast to earlier approaches, we do not need any external supporting infrastructure to be operational. Instead, the Smart Peer Group provides all needed functionality based solely on the resources available to the devices participating in the group.

The overall idea is that the devices present in a user's physical environment configure themselves automatically into a Smart Peer Group and allow the user to execute distributed applications with them. As an example, a presentation application could use a projector available in a lecture hall to output its data, while the application is controlled using a PDA. Furthermore, a facility management application could dynamically detect home automation actuators embedded in the environment and access them to switch off the lights for the presentation.

In more detail, we define a Smart Peer Group as a set of devices that fulfills the following properties:

Networking The nodes are networked using one or more short range wireless communication technologies. Although wired communication could be used, this would counter the ability of interacting with every device in the vicinity without any user interaction. Instead, the user would have to manually connect devices to the network by plugging them into some network hub. In addition, the user would need to reconnect devices to new hubs while moving around. Depending on the availability of a multi-hop routing protocol, devices may be able to communicate only with devices in their communication range, or with devices that can be reached over one or more intermediate nodes. However, the availability of such a routing protocol is not a necessary precondition for a Smart Peer Group in general.

Specialized Devices In Pervasive Computing a multitude of devices are embedded into everyday items [Wei91]. Due to the resulting restrictions on size, weight and cost, such devices are usually highly specialized and offer only a distinct set of capabilities. As an example, a smart pen is specialized for hand writing recognition to input data into a system. Its output capabilities however are highly restricted or not available at all. Note, that a Smart Peer Group may also contain general purpose computers. Therefore a Smart Peer Group usually contains specialized as well as general purpose devices.

Self-organization Self-organization in this context means that the user does not need to configure the system or single devices to enable their usage in the Smart Peer Group. Instead, the Smart Peer Group automatically detects new devices and integrates them into itself. This enables the development of truly unobtrusive Pervasive Computing systems, as envisioned by Mark Weiser [Wei91]. Otherwise, a system with lots of devices may become very distracting for the user. In addition, lots of devices in Pervasive Computing are expected to be embedded into everyday objects and are not experienced by the user as computing devices. Therefore, the user cannot be expected to manually configure them.

Peer-based Cooperation A Smart Peer Group relies solely on the spontaneous cooperation of the interconnected peer devices without support from any external infrastructure. Devices can directly access and use each others functionalities as long as they can communicate with each other. This is one of the main characteristica of a Smart Peer Group.

User Proximity Although multi-hop routing can be used to access remote devices, users are typically more interested in information and services in their physical proximity, e.g., for in-/output services or due to communication delay and energy consumption. Therefore, Smart Peer Groups concentrate on cooperation of nearby devices. However, cooperation with far away devices may still be possible, depending on the availability of a suitable communication link or routing protocol.

Shared Resources Most applications in Pervasive Computing will be distributed. The same is true for Smart Peer Groups. In fact, due to the missing external infrastructure with their powerful general purpose servers, even more applications may be distributed to be usable with a larger variety of different devices. To execute distributed applications, devices in a Smart Peer Group offer their functionalities, like e.g., a display or a data storage, to other devices in the Smart Peer Group. In a navigation system the user's mobile phone could offer his address list, while his PDA could provide access to its display.

From a technical point of view, a Smart Peer Group can be seen as a communication network partition in which all devices execute a special Smart Peer Group software that enables automatic peer-based cooperation between them.

As an example for a Smart Peer Group, take a smart jacket, a portable storage device and wireless earphones that are carried by a single user. The smart jacket offers the capability to input data using a set of keys embedded into its left sleeve. The storage device allows to save and access large amounts of data, e.g., music files. The wireless earphones output sound, e.g., music or text-to-speech messages. When the user starts a music player, the devices automatically connect with each other and cooperatively execute the player. On the other hand, a Bluetooth piconet by itself is not a Smart Peer Group, since the user has to manually integrate each device into the network.

2.2 Main Challenges

The properties of Smart Peer Groups given before result in a number of challenges. The three most important challenges are heterogeneity, dynamism and resource constraintness. In the following, we describe each of these challenges in more detail.

2.2.1 Heterogeneity

In Smart Peer Groups a large number of heterogeneous devices must be able to interoperate with each other seamlessly. Devices range from small embedded sensors to classic stand-alone computers. Clearly, the resources and capabilities, concerning e.g., CPU power or graphical output abilities, of such devices differ widely. Furthermore, devices from different manufacturers must be compatible and must stay so over multiple product cycles. As an example, a smart jacket manufactured by a given company must be able to cooperate with an electronically enhanced carpet produced by another manufacturer several years before. Otherwise product life times could be lessened unacceptably.

In addition, the devices that are present and usable in any given situation may be unpredictable, resulting in very heterogeneous execution environments. In order to work properly in every situation, applications must be able to function using a multitude of different device combinations. As an example, an application may output its data on a nearby large display if possible but may be able to use speech synthesis to output the data as audio if no display can be found.

Finally, communication between devices may be very heterogeneous. Devices are connected using wireless ad hoc networking technologies. The communication technologies used are highly heterogeneous ranging from infrared communication like IrDA to different radio-based technologies like IEEE 802.11 or Bluetooth. Interoperability protocols are tailored to specific requirements as well, e.g., a sensor does not need to implement a complex interoperability protocol but can simply emit its data periodically as events.

2.2.2 Dynamism

Due to device mobility the system is not static. New devices may enter the system at any time, while currently available devices may leave without warning. Network related properties like communication cost and bandwidth can change dynamically, e.g., because a device has moved out of communication range for IrDA but not IEEE 802.11. In addition, the functionality provided by a device may change even if the device itself is still available. As an example, an integrated GPS receiver may stop functioning when the device enters a building. Therefore, an application can never assume the presence of a given device or service even if it was available shortly before. Whenever the available devices change, applications may need to adapt themselves to the changes, e.g., because a used device becomes unavailable or because a better device has become available. Applications have to detect such changes and react accordingly.

However, system dynamism is not homogeneous. Instead, highly mobile devices may coexist with stationary devices or devices with similar mobility patterns, e.g., devices carried by the same user. Therefore, there may be stable parts in the topology that can be exploited by the system, e.g., by specifically combining devices to work together that have a small relative mobility.

2.2.3 Resource Constraintness

A large number of devices in Smart Peer Groups are highly resource limited concerning, e.g., computing power, memory, or energy. This is due to a number of factors. First, devices must be small, light, and cheap to be embedded into everyday items. Second, devices are often mobile and therefore battery-powered. Third, devices are always activated, because the user cannot be expected to manually activate them on demand. Instead, the system itself detects which devices are needed.

To make the situation even worse, Smart Peer Groups rely only on the resources that are available to the devices in the group. There is no dedicated external infrastructure that could be used to relocate resource intensive tasks to. Therefore the system must be able to operate on highly limited resources, e.g., if the group consists of a number of small sensor devices.

Clearly, not all devices found in a Smart Peer Group experience this problem. However, the system must be designed in a way to include all kinds of devices, ranging from resource rich stationary compute servers to resource poor mobile sensors.

2.3 Requirements

After discussing the properties and challenges of Smart Peer Group-based Pervasive Computing systems, we can now derive the requirements that such a system must fulfill [BS03].

2.3.1 Dynamic Device Lookup

As described before, Smart Peer Groups are highly dynamic. At any time, new devices may become available while existing ones may be lost. In addition, the devices's functionalities may change over time. To cope with this, the system must be able to continuously discover the currently available devices and their functionality, i.e., it must provide the means to discover new and leaving devices. This information must be forwarded to the application to allow it to adapt itself to the changed situation.

2.3.2 Adaptive and Interoperable Communication Support

After devices and their functionalities are detected, a suitable communication support must be provided to allow applications to access and use the remote functionality. As with the device lookup, the communication support must cope with the system dynamics. The availability of different communication technologies may change frequently and unexpectedly, e.g., because the user moves out of communication range. If a device offers more than one communication interface, it may switch between them to counter this. In addition, different communication partners may need different communication protocols. As an example, to access a legacy system using CORBA, the system should switch to IIOP. To interact with a device that offers web services, the system should switch to SOAP.

2.3.3 Decentralized Operation

Smart Environments can make use of central coordinating entities, e.g., to manage the knowledge about the capabilities of the environment. Thus, they can for instance rely on a central registry running on a well known device that is connected to the power grid. In Smart Peer Groups, the availability of such a device cannot be guaranteed. Instead, to be usable in the highly dynamic networks described before, the system must work despite frequent and unpredictable network partitions. It cannot rely on centralized components and must operate in a completely decentralized manner.

2.3.4 Configurability

To be usable on all kinds of devices found in Pervasive Computing, the system has to be tailorable to the device at hand, a sensor device as well as a mainframe. The core functionality should be small enough to be executed on a sensor platform, but easily extensible to use the capabilities of resource richer devices.

2.3.5 Energy-efficiency

Many devices in our system model are battery-operated. For such devices, the efficient usage of their scarce energy resources is crucial. Otherwise, device lifetimes may decrease significantly and the system's administrative overhead, e.g., to change batteries, increases. A typical electronic system can be structured into three main subsystems: the hardware platform, the system software, and applications. Energy-efficiency crosscuts this subsystems and must be addressed in all of them [BdM00]. Therefore, our infrastructure must be carefully designed to allow such energy-efficient operation and to co-exist with strategies on other system levels.

2.4 System Model

In the following we define the system model used in the remainder of this work (see Figure 2.1). We start by describing how to model devices in our system. After that we discuss how these devices are networked. Finally, we describe our assumptions concerning the user's behavior.

2.4.1 Device Model

As stated above, devices – or nodes – may range from sensor nodes over specialized systems to full fledged computers and mobile devices. For simplicity, we do not distinguish between stationary and mobile nodes. Instead, we restrict our system to so-called ultra-mobile devices [SBG99], i.e., mobile devices that can be used while moving.

Each node contains at least a CPU, memory, one or more bi-directional wireless communication interfaces, a battery, and a node-internal timer, e.g., a watchdog timer. In addition, nodes may contain additional hardware components that allow for further functionalities, e.g., a display or sensors.

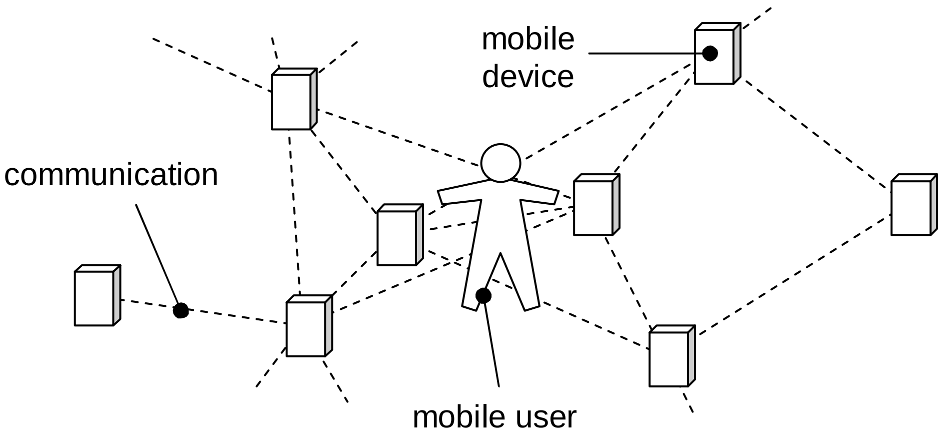


Figure 2.1: System model

As nodes are typically embedded into everyday items, the user may not be aware of the fact that he is currently using them. In addition, nodes may be located in inaccessible locations, e.g., at the ceiling. Therefore, nodes cannot be manually activated by their users but must be operational at any time. Each node is therefore assumed to be on and running at all time, until its battery is depleted. In this case, the node silently dies.

Nodes offer two modes of operation: sleep and awake, as shown in the power state machine [BHS98] given in Figure 2.2. The variables used in Figure 2.2 are explained in Table 2.1. Both modes consume a certain amount of power with $P_{sleep} << P_{awake}$. While in sleep mode, we assume that nodes cannot communicate or perform computations. The state of a sleeping node is preserved, i.e., the node can continue its operation after wakeup. Mode transitions demand a certain amount of energy and induce a transition latency. Transition from awake to sleep mode is initiated by a system call. Transition from sleep to awake mode is initiated by the node-internal timer. We assume no special hardware for waking up sleeping nodes remotely, e.g., by sending a signal [SBS02]. Each node offers a physical clock, either realized in hardware or software, that can be used to wait for a given time. Node clocks do not have to be synchronized.

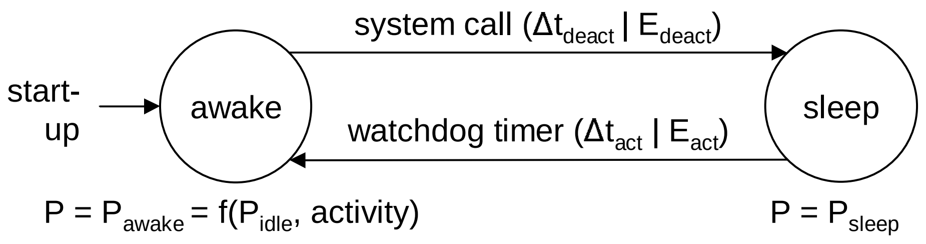

system call (Δt_{deact} | E_{deact})
start-up
awake
watchdog timer (Δt_{act} | E_{act})
sleep
$P = P_{awake} = f(P_{idle}, \text{activity})$
$P = P_{sleep}$

Figure 2.2: Power State Machine of a Node

However, we assume a bounded clock drift to be able to specify a node's remaining sleep time towards other nodes. Each node is identifiable by a globally unique ID that is imprinted in the node prior to system startup. We assume a certain operating system support on all nodes, offering access to the nodes current battery level, the system call to put the node to sleep mode, configuration of the internal wakeup timer, the node's clock and its ID.

Designator	**Description**
Δt_{act}	The time needed to activate a node.
Δt_{deact}	The time needed to deactivate a node.
E_{act}	The energy needed to activate a node.
E_{deact}	The energy needed to deactivate a node.
P_{awake}	The power consumed by a node while in *awake* mode. P_{awake} is a function of the idle power consumption P_{idle} and the activity of the node, e.g., the amount of data send and received.
P_{idle}	The power consumed by a node for idle standby.
P_{sleep}	The power consumed by a node while in *sleep* mode.

Table 2.1: Parameters used for the energy model

2.4.2 Network Model

The wireless connections between nodes differ with respect to the underlying technology and their characteristics. The most profound difference to classical computing environments is the spontaneous nature of such networks, which are formed by nodes that are temporarily in each others communication range. Obstacles, user mobility, and power saving are common events which lead to reconfigurations of the spontaneous network. Since nodes can be equipped with different network interfaces, spontaneous networks will overlap, i.e., some devices might be reachable via more than one network interface at a time. We assume communication technology with symmetric communication ranges and bandwidth. The device mobility may lead to frequent and unpredictable network partitions.

2.4.3 User Model

Our user model is rather generic. Users are mobile, carry one or more devices with them and access the system in order to execute applications for them. We assume users to be cooperative, i.e., they are willing to offer services to other user's clients. Users are unpredictable, i.e., it is not possible to reliably predict the future behavior of a user in the system, e.g., his movements. We also assume that all users are authorized to use all devices and all services. That means that there are no access restrictions, or that additional operations to grant access rights to users exist.

2.5 Summary

The focus of this work is in providing cooperation support for Smart Peer Groups. To do so, in this section we analyzed the main challenges of Smart Peer Groups and derived the requirements for such systems. Finally, we provided a system model that will be used throughout the rest of this dissertation. In the next chapter, we design a communication middleware tailored toward Smart Peer Groups. This middleware builds upon different communication mechanisms to allow devices to offer and access remote functionality. Furthermore, in Chapter 4 we provide a discovery system to dynamically detect the usable devices and their functionalities.

3

A Communication Middleware for Smart Peer Groups

The development of a Smart Peer Group based Pervasive Computing system that fulfills the requirements given before is a complex and error-prone task, leading to long development times and high costs. In order to ease this development task, we developed a number of concepts and algorithms for providing automatic cooperation between devices and integrated them into a Smart Peer Group system software. In this chapter, we present the basic building block of our system software, a communication middleware for Smart Peer Groups [BSGR03] [HBS03]. We named this communication middleware *BASE* because it serves as a foundation for applications as well as higher level adaptation support mechanisms. Key features of BASE are the uniform access to remote services and device-specific capabilities, the decoupling of the application communication model from the underlying interoperability protocols, and its dynamic extensibility supporting the range of devices from sensors [BBHS03] to full-fledged computers.

We start our description by presenting the design rationale taken in the development of BASE with respect to the requirements introduced before. After that, we discuss the middleware's architecture. We give some details about the implementation of BASE in Section 3.3 and discuss related work in Section 3.4. Finally, we give a short conclusion of the chapter.

3.1 Design Rationale

Existing middleware platforms are characterized by their precautions to overcome heterogeneity of computer systems with respect to the hardware platforms and programming languages. However, the computer systems on which applications are executed are mostly homogeneous according to their processing and storage capabilities. As already discussed, this is not the case for Smart Peer Groups. Instead, devices might be highly heterogeneous with respect to their resources and capabilities, including communication. In addition, network topologies will frequently change due to user and device mobility. The availability of resources, remote ones as well as local ones, can change over time, due to network connectivity as well as sensor-specific properties. A GPS sensor is not likely to work within a building and at night sensors based on daylight will stop operating. To cope with such ever-changing environments, a middleware must be highly flexible and configurable. In the following, we motivate the design of our middleware.

3.1.1 Uniform Programming Interface

Devices may provide different functionalities, e.g., temperature or positioning sensors, display or input capabilities, actuators to dim the light or to adjust the blinds of a window. To offer such functionalities to remote devices, they are modeled as *services*. A service specifies a well-defined *service interface*, that can be used by clients to access and use the service [GB01]. The availability of a service can be restricted in space and time, both locally and remotely. If a local service becomes unavailable, its clients may need to switch to a remote service dynamically and vice versa. To allow this, a uniform programming interface for accessing local and remote services is needed, i.e., our middleware has to provide access transparency for service usage. However, this does not imply location transparency. In some cases, programmers need to be aware of the current location of services. As an example, a programmer might prefer to use a local service due to the application's delay constraints. Therefore, our interface offers access transparency but allows to obtain additional information about services, e.g., their location.

3.1.2 Communication Model Decoupling

The communication model of a middleware, e.g., remote procedure calls (RPCs) or events, is typically reflected in its underlying interoperability protocol, e.g., using request/response messages or emitting event messages. This binds the useable interoperability protocols to the communication model used by the application. To select more flexibly between different protocols, we decouple the communication model of the application from that of the interoperability protocol.

This allows to abstract from any particular interoperability protocol when developing an application. As an example, an application can use a CORBA service using IIOP as well as a web service via SOAP without explicitly knowing about it. In addition we can use different communication links for outgoing and incoming messages. As an example, the middleware could send out a request as an event via infrared and receive the reply over an RPC interoperability protocol based on TCP and IEEE 802.11.

3.1.3 Strategy-based Communication Adaptation

If devices can communicate using multiple communication technologies and protocols, a suitable protocol stack has to be selected. To stay independent of any particular technology or protocol, this selection should be done by the middleware automatically. In addition, the middleware should reselect the used protocols at runtime, if needed. As an example, if communication over infrared becomes unavailable, the middleware could switch to radio-based communication automatically. To realize this, we offer a simple interface for the application developer to specify the application dependent communication constraints. Using this interface, a developer might specify, that the middleware should try to minimize the energy consumption for the communication or that it should encrypt communication. An adaptation strategy reads this specification and selects a protocol stack accordingly.

Depending on the desired flexibility of the selection process, different adaptation strategies exist. As an example, a static ordering of protocols and technologies can be specified at compile- or deployment-time. At runtime, the strategy simply selects the first protocol and technology that is available on both interacting devices. Alternatively, the strategy can try to determine the best combination dynamically using information collected at runtime. Such information can for instance include the current signal strength, the bandwidth, or the amount of exchanged data.

Both aforementioned strategies have their benefits and drawbacks. A purely static solution can be evaluated efficiently at runtime, but it might lead to suboptimal solutions in cases where the predefined ordering does not reflect the actual ordering. In contrast to that, a dynamic and completely generic solution might lead to inefficiencies during the selection process. Since protocol and technology selection is a task that must be performed frequently, especially if the properties of technologies are dynamic, these inefficiencies can easily lead to an intolerable performance. To support all different kinds of devices with different resource restrictions, the specific strategy that is used can be selected for each device. This allows resource-rich devices with many different communication protocols to execute a complex strategy, while resource-poor devices with only one or two possible protocol stacks can use a simple and very efficient strategy.

3.1.4 Micro-broker-based Architecture

As mentioned before, Smart Peer Groups contain a large number of very different devices, concerning the device's resources and capabilities. To be usable for all these devices, a middleware must be tailored toward the specific device properties. One way to do so is to provide a number of preconfigured middleware variants. As an example, CORBA specifies a minimal CORBA variant for resource poor devices [Obj02]. However, this could lead to a large number of variants that must be maintained simultaneously. In addition, the optimal middleware configuration may change at runtime and must be adapted dynamically.

Therefore we chose an extensible and configurable micro-broker-based architecture, that can be adapted to the current system environment at runtime. This approach is inspired by microkernels as they were introduced into the realm of operating systems (e.g., [RJO$^+$89], [TKvRB91]) and had some first applications in the middleware area as well (e.g., [PR00], [RKC01]). Only minimal functionality, i.e., accepting and dispatching requests, is located in the micro-broker. Interoperability protocols as well as object life cycle management can be added as additional services, realized as plugins. The benefit of our micro-broker approach compared to existing middleware platforms is the minimal footprint needed for a basic configuration, which qualifies it for small embedded systems as well as the extensibility providing the means to use features of more sophisticated computers.

3.1.5 Device Deactivation

In the past, a multitude of different approaches toward energy-efficient systems have been proposed. Following Flinn [Fli01], approaches can be classified into lower level approaches that focus on the design of low power hardware components, e.g., [McK00] [WKY04] [CSC02], and higher level approaches which operate on the operating system, e.g., [LCS$^+$00], [Ell99], or the application level, e.g., [TMWTC96]. Typically, the goal is to reduce energy consumption as much as possible while meeting a given performance goal, e.g., in terms of runtime [WWDS94] or application data quality [FS99].

Our approach toward energy-efficiency is based on two observations: first, in typical Smart Peer Groups, devices are often idle and wait to be used. This is the case, because most devices are never turned off. Doing so would require the user to manually switch them on again when they are needed again. Due to the large number of devices and the fact that the user might not be aware of their presence, this is not feasible. In addition, devices are typically specialized for a given task, e.g., sensing the temperature. Therefore, they are only used if the application that is currently executed needs their specific functionality. Otherwise, they are idle.

The second important observation is, that idle devices consume a considerable amount of energy [FN01] [CJBM02] [SBS02]. To reduce the consumption of such devices, special low-power modes with reduced functionality are often used. Depending on the current situation, devices switch between high-performance and low-power modes and can save a considerable amount of energy with little impact on system performance [LS98]. Examples include the deactivation of displays and hard discs or dynamic power scaling techniques used for central processors. Another very energy consuming component is the network adapter. Various adapters that are currently in use have a relatively high energy-consumption even while they are solely waiting for incoming messages. As an example, an Orinoco Gold IEEE 802.11b adapter requires around 800 mW in idle mode [FN01] as shown in Table 3.1.

Mode	Power consumption P
sleep	60 mW
idle	805 mW
send	1400 mW
receive	950 mW

Table 3.1: Power consumption of an Orinoco Gold IEEE 802.11b adapter

In general, transmission costs can be divided into three parts: sending costs, reception costs and operational idle costs. The first two depend on the amount of transmitted data and the energy cost per byte, the third on the system run time and the continuous energy consumption of the network adapter. Therefore, we can distinguish three possible approaches, to lower the energy cost of our system: First, we can lower the energy cost per send/received byte. To do so, a multitude of different approaches on all network layers have been proposed in the literature, e.g., energy efficient medium access control [SR98] [CSA99] or routing [WSR98]. An overview is given in [JSAC01]. Second, we can minimize the amount of transferred data, e.g., by using space-saving formats or compression techniques to serialize descriptions [XLWN03] or by introducing data management techniques like replication or caching.

Third, we can lower the idle cost. As described before, idle costs can make up a substantial fraction of the total costs for communication. Depending on the scenario, they may even outnumber the energy consumed for actually sending and receiving data [SK97]. To illustrate this, we present a simple analytical examination. We compare the cost to send and receive data to that consumed while being idle using the power consumption values shown in Table 3.1.

We model the potential savings of deactivating the network adapter for t seconds as:

$$E_{save}(t) = t * (P_{idle} - P_{sleep}) \tag{3.1}$$

and the energy consumed to send s byte of data at transmission data rate d_{tx} as:

$$E_{comm}(s) = \frac{s}{d_{tx} * (P_{send} - P_{idle})} \tag{3.2}$$

Formula 3.1 and 3.2 lead to:

$$s = \frac{d_{tx} * t * (P_{send} - P_{idle})}{(P_{idle} - P_{sleep})} \tag{3.3}$$

Using Formula 3.3, we can calculate, that the amount of energy saved by deactivating this network adapter for one second is sufficient to send an amount of data to a neighboring node at a communication speed of 1 Mbps of approximately $\frac{1Mbps*(0.14W-0.805W)}{(0.805W-0.060W)} \approx$ 152 kByte.

Clearly, an energy-efficient system should combine all three approaches to achieve maximum energy savings. BASE allows to select communication interfaces depending on their energy cost per byte using the strategy-based selection described before. Higher layers in the protocol stack, e.g., a presentation layer, can be selected based on their overhead in terms of energy consumption or communication overhead. As an example, a XML-based presentation layer may be inferior to a XDR-based one with respect to communication overhead. Finally, compression algorithms can be included dynamically into the selected protocol stack to lower the amount of send data. In addition, we lower the idle consumption of devices by putting them in their energy-conserving sleep mode. The ideal case is that all nodes spend all idle time in sleep mode, maximizing the amount of energy saved compared with staying awake while being idle. A possible extension of this approach is to allow not only idle but also currently used devices to temporarily deactivate themselves. This allows to guarantee a certain lifetime for nodes [ZELV03]. However, deactivating a used node can result in high usage delays for its clients, which in turn consume energy while waiting for the node to be reactivated. Additionally, it may result in an undesirable system behavior for the user, who experiences slower execution of his applications. These drawbacks must be carefully weighted against the additional energy savings. This is beyond the scope of this work and subject to future research as well as the integration of further energy-efficient communication protocols and data management techniques to gain even higher savings in the future.

3.2 Architecture

Figure 3.1 depicts the overall architecture of BASE. It can be structured into three parts: the application part, the micro-broker, and the middleware extensions.

The *application part* comprises clients, services and system services, e.g., the service registry. In addition, the application layer provides two programming interfaces towards the middleware. Stubs and skeletons can be used to access the middleware using an RPC-based abstraction. Invocations offer a message-based interface. An invocation is composed of a source and a target address, an operation with parameters, and additional information concerning the handling of the invocation. Internally, stubs and skeletons communicate with the middleware via invocations, too.

The *micro-broker* is the central part of the system, consisting of the InvocationBroker, the PluginManager and two registries, namely the device and the object registry. The InvocationBroker accepts and mediates requests from the application layer represented as invocations. It dispatches the invocation to either a local service, a local device capability or a communication protocol stack, which transports the invocation to a remote device. The PluginManager is responsible for the extension of the middleware with additional functionality. The object registry

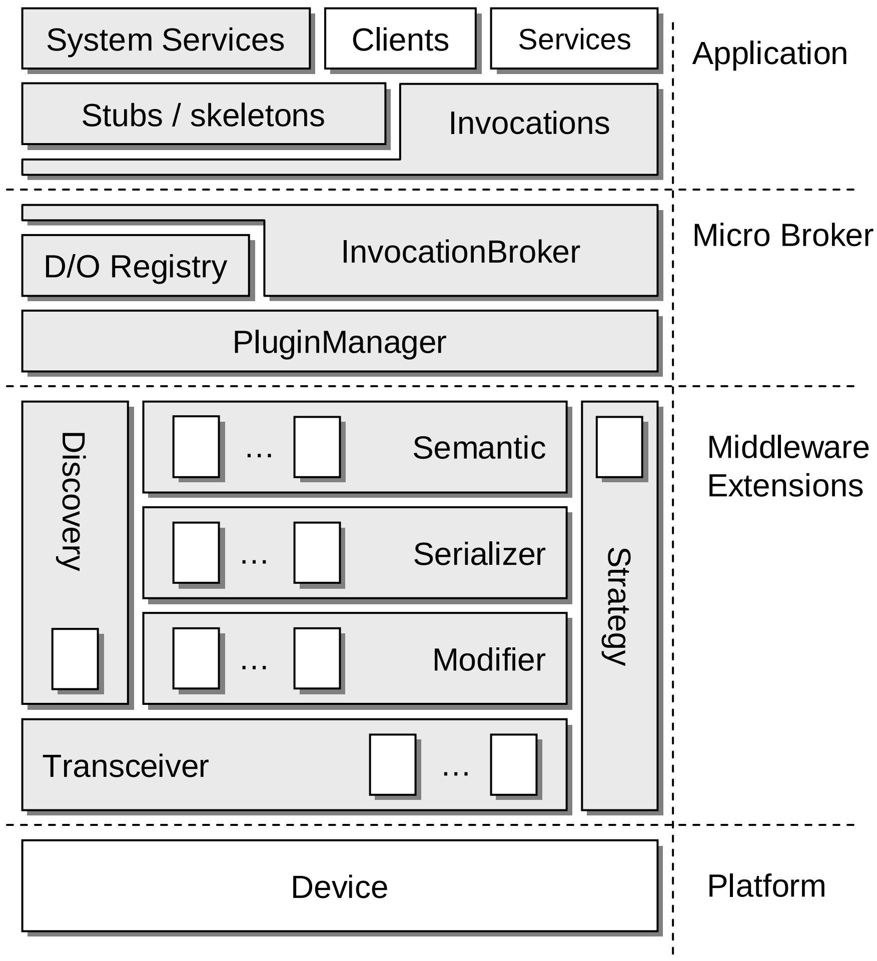


Figure 3.1: BASE architecture

contains all information necessary to mediate incoming invocations. The device registry stores all information necessary to access the currently available remote devices.

The *middleware extensions* consist of so-called plugins, which represent the entities capable of receiving invocations. Examples for plugins are transport protocols or encapsulations of device capabilities, such as sensor systems for positioning or temperature, or input/output capabilities such as printing or video projection. Plugins typically involve interaction with the underlying operating system or directly with the hardware to offer access to a device capability or transport. In addition, the extension layer contains the adaptation strategy used to select protocol stacks dynamically.

In the remainder of this section we discuss the individual parts of our middleware in more detail, starting at the middleware interface, i.e., invocations, and stubs and skeletons. After that, we present the InvocationBroker, the object registry, the device registry, and the PluginManager.

Finally, plugins and the adaptation strategy are described.

3.2.1 Invocations

Invocations resemble remote interactions in BASE. They carry requests from clients to services and back and are similar to dynamic invocation interface requests in CORBA. Figure 3.2 shows the elements of an invocation. Naturally, an invocation is represented as an object. Device and service IDs are used to denote a sender and receiver of an invocation. Services are given locally unique IDs. This ID is combined with a unique device ID, the so-called *SystemID*, to form a globally unique ID. The message IDs are needed for synchronization issues and are described in the paragraph discussing the InvocationBroker. A service context field allows the specification of additional parameters that indicate properties relevant to the processing of the invocation in the middleware such as synchronization issues or Quality of Service parameters. Basically, the context is a name-value list where parameters can be added freely. The payload contains the operations and parameters. In the case of event-based communication no receiver needs to be specified and the operation denotes the event-type on which applications can subscribe. The parameters then carry additional information of the event. In point-to-point communication the operations and parameters are interpreted as a remote method invocation.

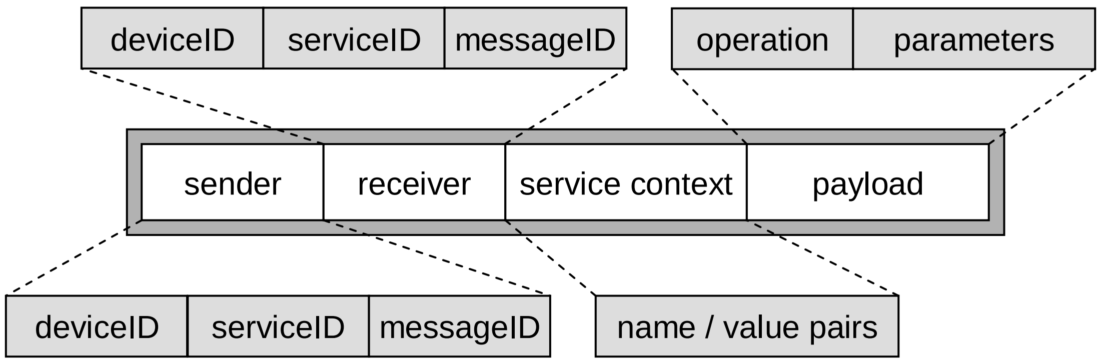


Figure 3.2: Invocation object structure

3.2.2 Stubs and Skeletons

In order to simplify the usage of our middleware, we provide a uniform interface for the different interaction types. Our goal is to provide a well known and easily usable abstraction for accessing local and remote functionality to application developers. We chose remote method invocation as this abstraction is widely known and very convenient. Our goal is to provide an easy interface but allow the programmer to easily configure the system.

A common abstraction in middleware systems are local proxies for remote entities providing local access for application objects – *stubs* representing the remote service to clients and *skeletons* issuing local calls to services. In BASE, stubs and skeletons rely on the invocation abstraction. Stubs generate invocations upon method calls, and skeletons generate local method calls upon a received invocation. Notice that the generation of an invocation does not result in the marshalling of the parameters. This is a responsibility of the transport plugins. Invocations are used here to provide a common concept for interaction with the micro-broker. Applications can, however, omit the use of stubs and skeletons and compose and interpret invocations directly.

In contrast to systems like Jini [Sun01], where stub and skeleton can include a service specific protocol stack this is not provided in BASE. Instead a service specific protocol would be realized as a plugin and thus become re-usable for other services as well.

3.2.3 Communication Models

To allow support for different communication models, BASE offers an interface for blocking and nonblocking transmissions, as well as future objects. In addition, different levels of reliability – e.g., at most once – and different message patterns – e.g., request/response – can be selected at the middleware. To do so, the caller includes information about its required reliability and message pattern in the service context field of the Invocation object and hands the Invocation to the middleware using a blocking or nonblocking call. The middleware then tries to realize the requested call by selecting appropriate plugins, e.g., a semantic plugin for a reliable request/response interaction, as discussed later.

Currently, BASE includes support for RPC and event-based communication. When using RPC, the caller can choose between a synchronous, asynchronous and deferred synchronous call. To support events, BASE includes a simple event service that handles all event messages given to the middleware and allows recipients to register for different event types.

Note, that the communication model provided by BASE to the application does not necessarily reflect the communication model of the interoperability protocol used to execute the call. BASE decouples these models to allow more flexible protocol selection, as described before. To do so, it manages an own thread-pool to decouple the calling application from the executing protocol. This is discussed in more detail in the next section.

3.2.4 InvocationBroker

The InvocationBroker provides the core functionality of the micro-broker. Invocations are accepted and dispatched. In order to separate the control flow between application and the processing of an invocation in a plugin, a thread pool is maintained. Incoming calls are entered into the invocation table, assigned a message ID in order to identify parallel invocations of the same client. The context field contains, among other information, the communication model, i.e., synchronicity and transactional pattern (request/response/event) of the invocation. Depending on the communication model, the InvocationBroker blocks the incoming thread in case of a synchronous invocation. A new thread from the thread-pool is taken and the delivery of the invocation to the responsible plugin (see below) is executed. After the plugin has processed the invocation by either a local action, e.g., retrieving a sensor data, or a remote action, i.e., marshalling and sending the request to a remote peer, the thread returns and is added to the threadpool again. In case of a remote processing, an invocation may be sent back to the initial caller. The invocation broker receives the invocation from a plugin for remote interaction, which may be different from the one that has processed the outgoing invocation, as shown in Figure 3.3. This allows the system to switch between different communication technologies even for currently running interactions and thus provides a high level of flexibility.

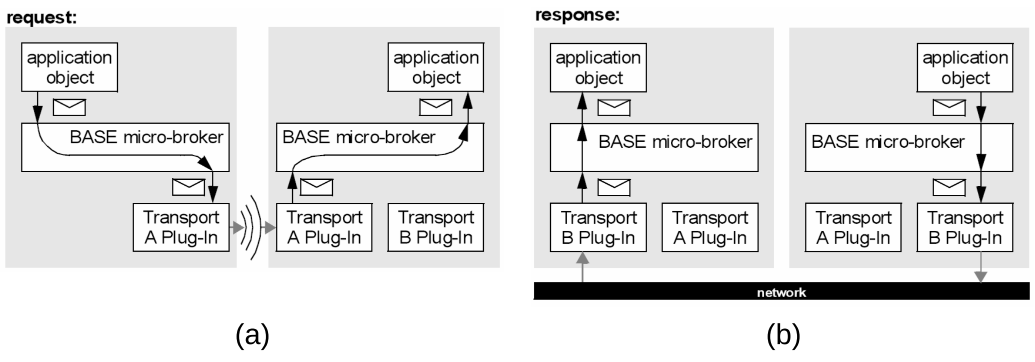


Figure 3.3: Request / response in BASE

The invocation carries the target object and its message ID. If a message ID is contained in the receiver field of the invocation, this indicates that a caller is either blocked or awaiting an asynchronous delivery of the invocation. In case of a blocked call, the waiting thread is continued and the invocation is provided as return. In the asynchronous case, the InvocationBroker takes a thread from the thread-pool and calls the application via a callback. In this case, the message ID is used to indicate the application callback registered at the InvocationBroker. Notice that the explicit handling of synchronization depending on the communication model retrieved from the service context is a major design decision in BASE. This decouples the communication model from the underlying interoperability protocols. A request/response based communication model can be realized over two event-protocols as well as an event can be sent as a single request in an RPC-based interoperability protocol. An interaction can take place over different plugins for out-going and incoming invocations. So far, BASE supports a limited number of communication models, but an extension to different synchronization models, see e.g., [Obj98], can easily be established with the underlying concept. In order to determine the target of an invocation or to provide applications with service lookup two registries are maintained and described below.

3.2.5 Service and Device Registry

The service registry records all locally available services on a device. Services – as mentioned before – can be either application objects offering a service or device capabilities. Applications can query for available services by either specifying a name (white pages) or the functional properties, i.e., the interface (yellow pages). Hence, a simple name and trading service is provided. Due to the nature of spontaneous networks, the availability of a lookup service cannot be assumed. The device registry maintains a list of all currently reachable devices and the plugins useable to access them. This allows for a simple service lookup in the vicinity of a device. If a service request cannot be fulfilled locally, registries of nearby devices can be queried and the result can be presented to the application. The information of the device registry is also used to determine which plugins should be used to contact a remote device. First, without any further information, any of the available plugins can be used. As long as there is a connection between two devices, i.e., the device is listed in the device registry and at least one communication plugin is provided, invocations can be exchanged. The service context sent with an invocation can be used to control the selection of specific plugins, e.g., in order to save energy or require a distinct bandwidth.

3.2.6 PluginManager

The PluginManager is essential for the abstraction BASE presents to an application developer. Platform-specific capabilities, e.g., device capabilities and transports, are represented as plugins and become accessible to the application programmer as services. The PluginManager allows the dynamic loading and integration of new plugins. Device capabilities are registered as local services. Communication protocols are registered at the InvocationBroker itself. Plugins provide an abstraction of device-specific resources. Depending on the platform interface that allows accessing the device capabilities, they can be portable among devices. Thus, an application on top of BASE will only interact via invocations, either dynamically constructed or generated by stubs, with device-specific capabilities. Plugins are responsible for accepting an invocation, marshalling it, and transmitting it as a protocol data unit to a remote peer, which then constructs an invocation by demarshalling it. The simplest communication plugin would use object serialization to marshal an invocation into a byte-buffer and send the buffer via a transport protocol, e.g., TCP/IP. Other plugins could rely on existing interoperability protocols and marshal and represent the invocation accordingly, e.g., map it to a request-message in IIOP and marshal the parameter by CDR, which allows interoperability with CORBA-based systems.

As long as the context of an invocation does not require a distinct plugin, the InvocationBroker may use any plugin to send an invocation to a remote device.

As mentioned before, the device registry maintains a list of all currently available plugins to a specific device. To obtain this information, the device performs a device discovery for all available communication technologies, e.g., by listening to periodic announcements from other devices. Upon detecting each other, devices can exchange lists of their plugins, storing them in their local device registries. This device discovery is integrated into the service discovery system discussed in Chapter 4.

3.2.7 Plugin Selection Strategy

To allow the integration of a selection strategy for different protocols and communication technologies, the PluginManager offers a *strategy interface*. The selection strategy uses this interface to interact with the PluginManager, e.g., to obtain information about the currently available plugins or to contact them. The strategy is called whenever a new protocol stack is needed to communicate with another device. It selects a suitable protocol stack that is then – in a second phase – instantiated by the PluginManager.

In order to support application-specific optimization goals during the selection process, we provide a small framework that enables application developers to attach requirements to Invocation objects. This framework mainly consists of a collection of name-value pairs that allows the specification of ordered dimensions to control the selection strategy's behavior. With a collection, developers can for instance specify that they prefer a certain type of plugin over another type or that they rely on the usage of a specific plugin or technology. In addition, plugins can specify additional requirements or preferences on the selection process. This allows, e.g., a given plugin to specify that it must be used in combination with a second plugin to work correctly. Furthermore, each plugin offers a dynamic description of its current state to enable the cross-layer reuse of internal information about their state. This allows strategies to consider plugin specific information such as the current signal strength – e.g., to select the technology with the best current transmission quality – or the used compression rate.

To sum up, the protocol selection is based on the used selection strategy, specific requirements of the communicating application and/or the transferred data, the available plugins and their dependencies with each others, and the dynamic state and configuration of the plugins. This variety allows for a very flexible protocol selection.

To transfer data between all involved entities, the service context contained in the Invocation object as described in Section 3.2.1 is passed between them and my be modified by each participant, e.g., to add new requirements on the selected plugins.

Concerning the granularity of plugins, BASE offers different possibilities to plugin developers. A plugin can contain a complete protocol stack or parts of it. Such plugins enable developers to create highly efficient integrated protocol implementations. However, the available protocol stacks are restricted, due to memory restrictions. Smaller sized plugins offer more flexibility, as they can be combined at runtime to create different protocol stacks. However, they require a stricter layering to allow an unrestricted combination.

BASE contains a layering concept for plugins that enables the PluginManager to create protocol stacks dynamically by combining plugins working on different protocol layers. The basic layers that BASE supports are *Semantic*, *Serializer*, *Modifier*, and *Transceiver*. The Transceiver layer receives and sends data using the capabilities of the platform. The Modifier layer contains plugins which allow to modify marshalled Invocations, e.g., for compression or encryption purposes. In the Serializer layer, Invocation objects are marshalled and unmarshalled. Finally, the Semantic layer realizes different call semantics, such as at-most-once, or at-least-once. To do so the semantic layer interacts tightly with the InvocationBroker that performs the synchronization of Invocations. A complete protocol stack must at least include the Semantic, Serializer and Transceiver layers. In addition, the stack may contain an arbitrary number of plugins on the Modifier layer. To allow both dynamically combined protocol stacks as well as statically predefined stacks that are implemented in a single plugin, our design enables the usage of so-called multilayer plugins that encapsulate two or more layers within one plugin.

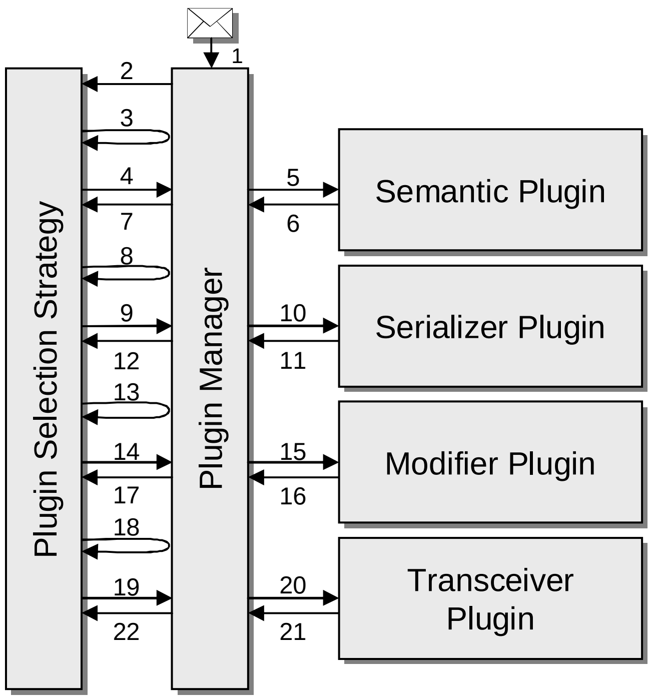


Figure 3.4: Plugin selection

The generic selection process is shown in Figure 3.4. First, the Invocation object containing the data to transmit, its destination, and the application's preferences included in the service context is passed to the PluginManager by the InvocationBroker (1). The PluginManager extracts the destination and the service context and forwards this information to the selection strategy (2). The strategy now selects a suitable semantic plugin. To do so, it requests a list of all semantic plugins that are compatible with the destination device from the PluginManager (3) and orders them according to its internal criteria and the application's preferences. Then, it calls the PluginManager again to contact the selected plugin and transfers the service context to it (4). The PluginManager calls the plugin with the given service context (5) and the plugin may add information about its current state as well as its own requirements to the context. After that, control is given back to the selection strategy (6-7) which processes the new information and repeats the cycle to select suitable serializer (8-12), modifier (13-17), and finally transceiver plugins (18-22).

Note that the selection process must only be performed by the sender, since the receiver has no other choice than using the initially selected plugins. However, since in BASE an RPC-like call is mapped to multiple invocation transfers, the used protocols might change within an interaction.

3.3 Implementation

Our middleware has been implemented in Java to rely on its platform-independence. Although, for small devices C or C++ would seem to be a better choice at first, we found that Java allows us to run our middleware on a multitude of different devices, if the used Java features, like reflection, etc. are carefully restricted. A Tini minicomputer [Loo01] for example can execute only a subset of Java Version 1.1. Other devices, like smart phones or PDAs are limited to the Java Microedition [Sun]. Thus, we defined a restricted subset of the Java features and class libraries that was used for the implementation. In [BSGR03] we measured the static and dynamic memory footprint of BASE and showed, that we are able to execute a minimal middleware installation with as little as 132 kByte.

Our implementation was successfully deployed and tested on embedded devices like the JStamp [Sys06] and the Tini minicomputer [Loo01], Microsoft Windows CE-based PDAs, Tablet PCs and Laptops, desktop computers working with Microsoft Windows and Linux, and Sun Solaris-based workstation computers. We implemented plugins for wired and wireless communication technologies, like IEEE 802.11, Bluetooth, and IrDA. BASE was used in the 3PC project as the communication middleware for the Pervasive Computing component system PCOM, and in the CupID project to access and control a networked coffee machine from WMF. In addition, BASE was demonstrated on a variety of different occasions, like PerCom 2006, the university's open house, or visits from other research groups.

3.4 Related Work

In the following section, we discuss related approaches for middleware systems. We specifically address well known conventional systems, dynamically reconfigurable middleware systems, and middleware for resource-poor devices. We conclude our discussion with a short summary.

3.4.1 Conventional Middleware Systems

Device heterogeneity is not a unique characteristic of pervasive computing, but can be found in conventional systems, too. Different middleware systems like CORBA [Obj04], Java RMI [Sun02] or DCOM [EE98] have been developed to provide a homogeneous access to remote entities independent of e.g. operating systems or hardware architectures. Typically, these middleware systems try to provide as much functionality as possible, which leads to very complex and resource consuming systems, that are not suitable for small devices. Approaches to solve this problem exist and are discussed below. Conventional middleware systems are designed for mostly stable network environments, in which service unavailability is a rare event and can be treated as an error.

3.4.2 Dynamically Reconfigurable Middleware

Extending conventional middleware systems to dynamically reconfigurable middleware systems (e.g. [Led99], [RKC99], [BG00], [BCRP00], [BCA$^+$01], [RKC01]) enables such middleware to adapt its behavior at runtime to different environments and application requirements, e.g.,how marshalling is done. Still, different communication models or different protocols for outgoing and incoming messages are typically not supported. As one exception, the Rover toolkit [JTK97] provides this functionality for its queued RPC (QRPC) concept, layered on top of different transport protocols. However, Rover only supports the QRPC and addresses potentially disconnected access to an infrastructure and not spontaneous networking. A further difference from BASE is that most existing reconfigurable middleware systems concentrate on powerful reconfiguration interfaces and not on supporting small, resource-poor devices. A notable exception to this is UIC [RKC01], which is discussed in the next section.

3.4.3 Middleware for Resource-Poor Devices

The resource restrictions on mobile devices prohibit the application of a full-fledged middleware system. One way to address this is to restrict existing systems and provide only a functional subset (e.g., [Obj02], [RSC$^+$99], [Sch04]) leading to different programming models or a subset of available interoperability protocols. Another option is to structure the middleware in multiple components, such that unnecessary functionality can be excluded from the middleware dynamically. One example is *the Universally Interoperable Core* (UIC) [RKC01]. UIC is based on a micro-kernel that can be dynamically extended to interact with different existing middleware solutions. Still, the used protocol stack is determined before the start of the interaction and cannot be switched between request and reply as in BASE. In addition, although the UIC micro-kernel can be adapted dynamically, UIC concentrates mostly on predefined, static configurations for legacy system support.

3.4.4 Summary

To sum up, middleware system have been developed in a multitude of system models and for a number of different requirements. However, conventional middleware systems are to inflexible and resource-intensive to be used in Smart Peer Groups. Other systems specialize on distinct requirements, e.g., configurability, but do not address all our requirements. BASE combines a highly flexible architecture that allows the middleware to be adapted to many different scenarios at runtime, with the ability to operate on very resource-constraint devices without relying on any external infrastructure support.

3.5 Conclusion

In this chapter we have presented the concepts and design of BASE, a flexible middleware supporting the requirements of pervasive computing environments. Based on a micro-broker design, BASE allows minimal installations on embedded devices or specialized platforms as well as the integration of features available on resource-rich devices, such as personal computers. Application programmers can rely on a uniform abstraction to access remote and local services as well as device-specific capabilities. Thus BASE supports the portability of applications across heterogeneous devices. The middleware shields applications from the multitude of different communication technologies and interoperability protocols by separating the communication model of the application and the interoperability protocols used. This allows the usage of nearly arbitrary interoperability protocols. Furthermore, BASE supports the adaptation to fluctuations in communication connectivity by switching dynamically between different communication technologies and associated protocols during interactions.

4

A Service Discovery System for Smart Peer Groups

Besides the ability to contact and use remote services, which is offered by our communication middleware BASE, applications need up-to-date information about which services are currently available. This includes information about the communication protocols needed to contact a service or about a service's current state. Using this information, the applications can select the services that are best-suited for their specific needs. The provision of this information is the responsibility of a service discovery system. To do so, the system exchanges descriptions of client needs and service offers between the participating devices.

In this chapter, we present a service discovery system, called *Service Awareness and Discovery in Mobile Ad-hoc Networks* (SANDMAN) [SBR04], which we developed specifically for Smart Peer Group-based Pervasive Computing systems. The chapter is structured as follows: first, we derive the objectives of our discovery system. Then we discuss the design rationale and present our approach SANDMAN. We discuss the architecture of SANDMAN and present its three subsystems. After that, we present a discussion of existing service discovery systems. Finally, we conclude the chapter with a short summary.

4.1 Discovery Objectives

In Section 2.3 we presented the requirements for Smart Peer Group-based Pervasive Computing systems in general. A service discovery system for Smart Peer Groups has to fulfill these requirements. In addition, there are a number of objectives that a service discovery system should fulfill. We identify three such objectives, namely the discovery latency, the discovery precision, and the discovery recall, which we discuss in the following.

4.1.1 Discovery Latency (l)

Applications in Smart Peer Groups aim for prompt discovery of devices and their services in order to plan and prepare for their usage. If a service that is currently used becomes unavailable, the client should detect a suitable replacement service as quickly as possible to continue its operation. Otherwise, the user may experience disturbances in the system execution. This is especially important, since planning a suitable application adaptation to react to the lost service is a complex task and may itself require some time. In addition, switching to another service may require that service to perform a service discovery itself, because the service may in turn use other services to operate. Thus, the system could experience multiple sequential discovery latencies. Therefore, a service discovery system should try to minimize the latency of discovery requests. Using t_{snd} and t_{rcv} described in Table 4.1 we define the *discovery latency* l as:

$$l := t_{rcv} - t_{snd} \tag{4.1}$$

The discovery latency should not be confused with the *usage latency*. The usage latency describes the time between the discovery of a service and its possible usage. As an example, if the discovered service is offered by a currently deactivated device, the client has to wait until the device is activated before using the service.

4.1.2 Discovery Precision (p)

In order to save system resources, a discovery system should report only results, that really match a given discovery query, i.e., services that can really be used by the client. To describe this system property we introduce the *discovery precision* p. The discovery precision corresponds to the notion of *precision* used in information retrieval (see, e.g., [Fer03]). Using the definitions given in Table 4.1 we define p as:

$$p := \frac{|\mathcal{S}^d_{v_i} \cap \mathcal{D}^d_{v_i}|}{|\mathcal{D}^d_{v_i}|} \tag{4.2}$$

Informally, p describes the fraction of correctly discovered services compared to the total number of discovered services. If a discovery request yields for example four services as its result but only two of them are correct, we have a precision of 0.5. Services may be discovered incorrectly e.g., if they are still announced after the node offering them has already left the Smart Peer Group. A precise discovery is important, because wrongly detected services may degenerate system performance. If a client chooses the incorrectly discovered service, it may wait for the offering node to awaken before trying to communicate with it and detecting the error.

Designator	Description
$\mathcal{S}$	the set of all services s
d	a list of service properties, e.g., name, type, that can be used to describe a service or a set of services.
$\mathcal{S}_{v_i}^d$	the set of all services $s \in \mathcal{S}$ that match the descriptor d and are available for clients executed on node v_i
$\mathcal{D}_{v_i}^d$	the set of all services $s \in \mathcal{S}$ that are returned as the result of a discovery query issued by a client executed on node v_i for services matching the descriptor d. Note, that this does not imply, that all $s \in \mathcal{D}_{v_i}^d$ match d
t_{snd}	the point in time where a discovery request is send by the client
t_{rcv}	the point in time where the result of a discovery request is received by the client

Table 4.1: Parameters used for describing service discovery objectives

4.1.3 Discovery Recall (r)

Complementary to discovery precision, the *discovery recall* r denotes how complete the discovery works, i.e., how many of the services that should be discovered are really discovered. As before, we derive this objective from a similar parameter – the recall – used in information retrieval systems (see, e.g., [Fer03]). A service discovery system must aim at providing a high discovery recall. Otherwise, clients may miss important services and may be unable to work efficiently or not at all. As an example, if a client misses the availability of a color display in the vicinity, it may unnecessarily choose to use a monochrome display or even an audio output instead. Again using the definitions given in Table 4.1 we define r as:

$$r := \frac{|\mathcal{S}_{v_i}^d \cap \mathcal{D}_{v_i}^d|}{|\mathcal{S}_{v_i}^d|} \tag{4.3}$$

As an example, if there are four services available,which match a discovery request but the system reports only two of them, we have a discovery recall of 0.5. A low recall could e.g., result from a highly mobile system where the discovery system has not yet discovered new devices and their services. In addition, if nodes are deactivated to save energy without further precautions, deactivated nodes may simply miss discovery requests, again resulting in a low recall.

4.1.4 Discussion

The three objectives discussed before aim at providing a high discovery quality. In the ideal case, a discovery system would provide a discovery with a latency $l = 0$ ms, a precision $p = 1.0$ and a recall $r = 1.0$. In practice, however, this case is hardly achievable, e.g., because of communication delays. In addition, the objectives are partly in contrast with the requirements discussed in Section 2.3. As an example, a system searching for new services only once every 30 minutes may be very energy-efficient but will result in a poor discovery recall. Continuously searching on the other hand may provide a high discovery recall but could deplete the node's batteries fast. Therefore, a concrete service discovery system has to provide a reasonable trade-off between these conflicting goals.

4.2 Design Rationale

As already mentioned, the goal of a service discovery system is to provide all clients with all information necessary to select and access services that are best-suited for their specific needs. To do so, the system exchanges descriptions of client needs and service offers between the participating devices.

4.2.1 System Organization

Service discovery systems can have a *mediator-based* or a *peer-based* system organization. In mediator-based systems (e.g., Jini [Sun01]), service discovery is performed using one or multiple mediators, called lookup services (LUS). Services are registered at the LUS and published by them. LUS can be independent from each other (like, e.g., in Jini), can cooperate (like, e.g., in UDDI [udd04]) or can form a hierarchy (like, e.g., in INS [AWSBL99]). In peer-based systems (e.g., UPnP [Mic00]), each service provider is responsible for publishing its own services without any support by a LUS. As an example, in UPnP nodes offering a certain service directly reply to clients searching such a service.

To distribute information about client needs and service offers between two nodes, both nodes have to be activated at the same time. Otherwise no communication is possible. To ensure this, two different approaches exist. First, all devices can synchronously deactivate and reactivate themselves in such a way that the devices have a well-known time interval that can be used for direct communication (*synchronization-based communication*). Second, the devices can select a third node that acts as a communication proxy between them. Whenever a node wants to send an information, it sends this information to the proxy device. The proxy stores it and waits until the destination node becomes active. At this time, the proxy node forwards the information to the destination node (*proxy-based communication*).

Peer-based and mediator-based system organization can be combined with synchronization-based and proxy-based communication, leading to four possible approaches as shown in Figure 4.1.

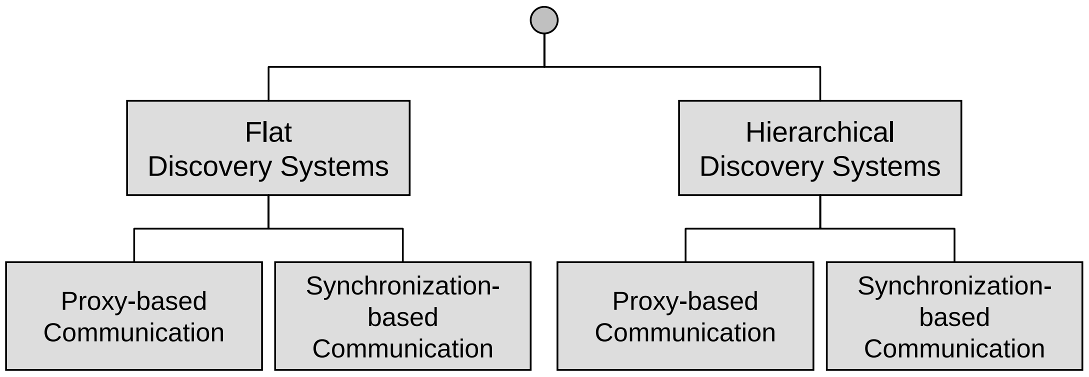

Flat Discovery Systems
Hierarchical Discovery Systems
Proxy-based Communication
Synchronization-based Communication
Proxy-based Communication
Synchronization-based Communication

Figure 4.1: Discovery system organization

Peer-based System Organization with Synchronization-based Communication

If we combine a peer-based discovery system with synchronization-based communication, nodes use synchronized wake times to exchange service requests and offers. To do so, messages are broadcast in the network. The main advantage of this approach is its simplicity. All nodes are equal and execute the same code. In addition, in networks, where the set of participating nodes is static, communication adapters can be disabled for a large percentage of the system life time. This is the case, because in such networks the nodes can synchronize themselves once at startup time. Unsynchronized new nodes do not need to be handled. In dynamic environments like Smart Peer Groups however, communication adapters can be disabled for a maximum of 50% of the time. If the period is longer, synchronization cannot reliably guarantee that an unsynchronized device will ever be found as the devices' wake times could never overlap. Furthermore, depending on the frequency of the synchronized wake times, the system may experience high discovery latencies. Therefore, this approach does not fulfill our requirements concerning energy-efficiency and discovery latencies.

Peer-based System Organization with Proxy-based Communication

In a peer-based discovery system with proxy-based communication, nodes communicate indirectly using proxy nodes. Service requests and offers are again exchanged using broadcasts. This approach can achieve higher deactivation percentages and therefore save more energy if the number of proxy nodes is small compared to the total number of nodes. However, due to the mobility of devices, proxies must be determined dynamically at runtime using suitable election protocols. If proxies are poorly chosen they must be reelected frequently. Since doing so consumes energy, the overhead for maintaining proxies can easily outweigh the potential savings in such cases. Therefore, proxy-based approaches may not be beneficial for highly dynamic environments. As discovery requests are temporarily stored at the proxies and answered by each node individually when activated, the discovery delay depends on the duration that nodes are deactivated. This again may lead to high discovery latencies. Therefore, this approach does not fulfill our requirements.

Mediator-based System Organization with Synchronization-based Communication

In a mediator-based discovery system with synchronization-based communication, discovery requests are directed to one or more LUSs. To do so, nodes synchronize their activation times with each other. Concerning discovery delays this approach behaves like a peer-based system with synchronization, again experiencing high discovery delays. In addition, this approach has to dynamically adapt which nodes act as a LUS and which LUS is used by a node. Otherwise, due to the node mobility, nodes may become disconnected from their LUS and high discovery inaccuracies may arise. This can happen, e.g., because a LUS announces a service that is no longer available (low precision) or misses others that are new in its vicinity (low recall). In highly dynamic system environments, this may lead to continuous adaptations which counter the potential energy savings. In summary, this approach combines the disadvantages of the first two approaches, namely low deactivation times and high discovery delays. Thus, it does not fulfill our requirements as well.

Mediator-based System Organization with Proxy-based Communication

Finally, a mediator-based discovery system that uses proxy-based communication can be designed. Here, client needs and service offers are distributed using a communication proxy and are send to a LUS which mediates the discovery process. Again, this system has to adapt itself at runtime, to make sure, that both communication proxies and LUS are distributed in the network as needed. However, if both functions are provided by the same nodes, the system can combine both adaptation processes and benefit from the resulting synergies. In addition, low discovery delays are achievable, independently from the length of the node's deactivation times. This is the case, because the proxy receiving the message is at the same time a LUS and can answer the request right away, without the need to wait for any node to be activated again. In summary, this forth approach is the only one that is able to fulfill all our requirements. It achieves deactivation periods larger than 50 % as well as small discovery delays.

Conclusion

We choose the forth approach, the mediator-based system organization with proxy-based communication, for our service discovery system. This approach is able to fulfill all our requirements. Its main drawback is the overhead resulting from system reorganizations due to the mobility of the devices. To limit this overhead, we restrict our approach to relatively stable system parts in terms of mobility. This allows to save energy in such environments while providing a high level of discovery precision and recall as well as a low discovery latency. In highly dynamic environments on the other hand, we choose to let nodes stay activated and operate in a peer-based system organization with direct communication. Otherwise, either precision and recall could degenerate unacceptably or the overhead resulting from constant and frequent system reorganizations could become very high.

4.2.2 Information Distribution Pattern

After deciding on the overall system organization of our discovery system, we must decide which distribution pattern should be used to distribute service-related information in the system. Service information can be either distributed proactively by sending service announcements or reactively by sending lookup requests that are answered by a suitable LUS.

To achieve high discovery precision and recall, a proactive approach has to send announcements frequently, as this determines how long a node that left communication range is still announced (precision) and how fast a new node is found (recall). This is true even if no client is currently searching a service offered by these nodes as the system can never be sure that no new client has started searching. If announcements are stored at all nodes, very low discovery latencies are achievable. However, resource-poor devices may not be able to store all service announcements locally due to memory restrictions. Such devices must wait for a full announcement cycle before they can answer a discovery request. This either leads to a high discovery latency or to frequent announcements and therefore high energy consumption.

A reactive information distribution pattern does not suffer from these disadvantages. In many cases, a client will be perfectly happy with its service once it has found one. Therefore, a new discovery request must be distributed in the system only if the client detects that the service has become unavailable and the client must choose another one. If a client continuously needs information about the available services, e.g., because it has not yet detected a suitable service, the system has to resend discovery requests frequently to achieve high precision and recall. However, each client can choose how often its request is resend independently and therefore tune the resulting energy consumption to its current needs. Due to these reasons, we choose a reactive distribution pattern where clients send discovery requests and receive response messages about possible service providers from LUSes.

4.3 SANDMAN

With the design decisions taken in the previous section, we can now present the overall design of our service discovery system *SANDMAN*. SANDMAN offers two different modes of operation. If the system environment is highly dynamic, SANDMAN keeps nodes activated to ensure, that they can be discovered accurately and with little latency. In such environments, a mediator-based system organization would result in a large overhead to maintain the system hierarchy. Therefore, the system uses a peer-based system organization in which all nodes are equal peers. Discovery requests are answered by each node directly.

However, if the system identifies a more stable network topology, it switches to a mediator-based system organization in which nodes are dynamically elected to operate as LUS for other nodes. This is realized by organizing the nodes dynamically into node clusters.

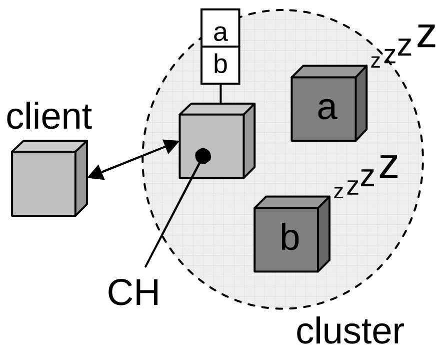


Figure 4.2: SANDMAN approach overview

As depicted in Figure 4.2, a cluster consists of one cluster head (CH) and an arbitrary number of clustered nodes (CN). Both, CH and CNs can offer and use services. CNs periodically deactivate themselves, e.g., if they are idle, in order to save energy. After waking up, a CN waits for incoming client requests for the duration of a timeout interval. If no requests are received, it informs its CH and deactivates itself again. This way, CNs that are not used continuously can save a substantial amount of energy. To provide timely discovery with low latencies despite the deactivated nodes, the CH stays active all the time. It acts as LUS managing all services running on nodes in its cluster. Therefore, the CH can instantly answer any client's discovery request with a suitable list of service descriptions and wake-up times of the respective CNs. Note, that each CN must check with the CH periodically to limit the resulting loss of discovery precision and recall. Otherwise, a CH could announce a service that is no longer available, e.g., because the node offering the service has left the CH communication range and the CH has not detected this. Letting the CH stay awake constantly is the trade-off taken in SANDMAN between energy-saving and discovery latencies.

4.3.1 System Structure

SANDMAN can be divided into three parts (see Figure 4.3): The first part, the *cluster management*, is responsible for creating and maintaining the clusters. It determines which nodes act as CH and assigns nodes to certain clusters. It also detects changes in node connectivity and adapts the system accordingly by re-electing CHs and re-assigning nodes. The second part, the *service management*, defines how services are registered at a cluster head and how clients perform lookups. The third part, the *energy management*, identifies idle nodes and schedules and executes their deactivation. These parts do not act completely independent but cooperate, e.g., in order to schedule sleep times depending on client requests or to forward client requests to appropriate services.

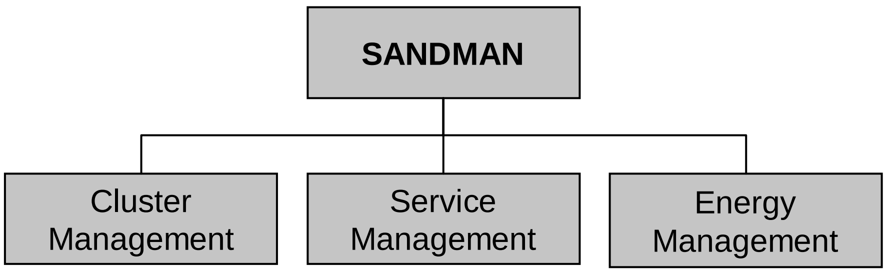


Figure 4.3: SANDMAN structure

In our system architecture as depicted in Figure 4.4, we model these three parts as distinct architectural entities: the *ClusterManager*, the *ServiceManager* and the *EnergyManager*. Each part is designed as a BASE system service. This allows them to use all BASE functionalities to conveniently access and use remote services, like, e.g., their counterparts on other nodes. Local interaction between the parts is provided by defining a set of operations and events that each part emits or catches. Although this interaction could have been realized using BASE mechanisms, too, we chose to rely on local events to get a higher performance. This is unproblematic because the three SANDMAN parts are integrated rather closely on each device and cannot be used without each other. In addition to the three system services, we provide a discovery plugin for SANDMAN, as required by BASE. This plugin is used to perform low level actions for SANDMAN, like, e.g., broadcasting alive messages to nodes in the vicinity or updating the BASE device registry if a new neighboring node is detected.

In the following chapters, we describe the three parts of SANDMAN in more detail, starting with the cluster management, the service management and finally the energy management. Before we do so, however, we present the roles that nodes can be assigned to and the basic communication primitives, that SANDMAN builds upon.

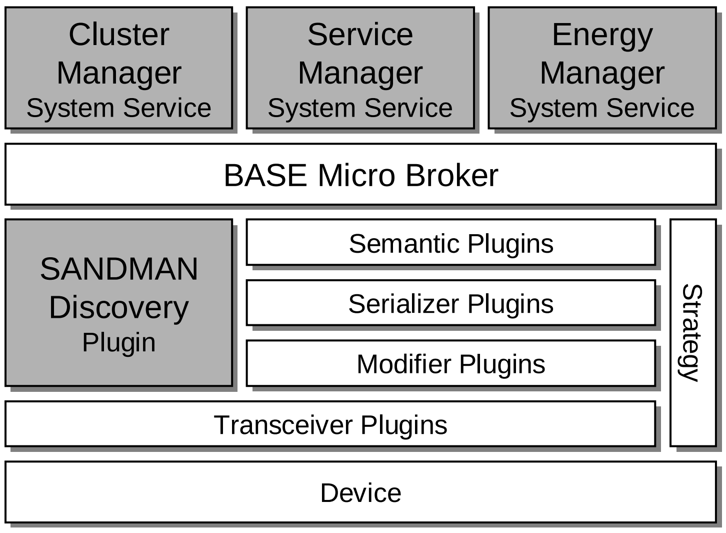


Figure 4.4: SANDMAN architecture

4.3.2 Node Roles

SANDMAN introduces three node roles: unclustered nodes, cluster heads and clustered nodes. These roles are defined in the following.

Definition 4.1 (Unclustered Node) *An* Unclustered Node *(UN) operates by itself and is not a member of any cluster. It performs all service discovery operations by itself and answers discovery requests from remote clients.*

Definition 4.2 (Cluster Head) *A* Cluster Head *(CH) is a member of exactly one cluster and acts as the leader of this cluster. There is exactly one CH in each cluster. The CH is responsible for service discovery operations for itself and all nodes in its cluster. It manages the service registry containing all services in the cluster and answers discovery requests from remote clients.*

Definition 4.3 (Clustered Node) *A* Clustered Node *(CN) is a member of exactly one cluster and not a CH. There is at least one CN in each cluster. An empty cluster leads to the CH switching back to the UN role. A CN has turned responsibility for all service discovery operations to its CH. Therefore it does not directly answer discovery requests and manages only its local services.*

4.3.3 Basic Communication Primitives

Before discussing SANDMAN in more detail, we present the basic communication primitives that SANDMAN builds upon. These primitives are realized by BASE. When a message is received, the receiving entity can identify the sender by accessing the *sender* variable, which contains the sender's SystemID.

One-hop Broadcast (bsend)

A node v_i may broadcast a data packet with best-effort semantics to all nodes $v_j..v_k$ in its one hop neighborhood using the `bsend` operation:

```
bsend(type, payload)
// in type    : message type
// in payload : data to send
```

This operation takes two parameters. The first denotes the type of the message to send (*type*). The second gives the data to send (*payload*).

Unicast (usend)

To communicate reliably with a single node, a node can use the `usend` operation:

```
usend(dest, type, payload)
// in dest    : receiver id
// in type    : message type
// in payload : data to send
```

This operation takes three parameters, the unique id of the receiver of the message (*dest*), the message type (*type*) and the message payload (*payload*). Using a multi-hop routing algorithm, `usend` can deliver messages to nodes that are multiple hops away.

Multicast (msend)

In addition to sending broadcasts and unicasts, a node may communicate with a group of nodes using a multi-hop multicast protocol. To send a message to a multicast group, a node can use the `msend` operation.

```
msend(group, type, payload)
// in group   : receiver group
// in type    : message type
// in payload : data to send
```

The `msend` operation takes three parameters, the group to send this message to (*group*), the message type (*type*) and the message payload (*payload*).

4.4 Related Work

After describing the basic design of our service discovery system, we present a discussion of existing service discovery systems in this section. We start with conventional discovery systems and specifically address approaches for resource poor devices and energy efficiency. Finally, we describe related approaches for energy efficient communication and conclude the section with a short summary.

4.4.1 Conventional Service Discovery Systems

There are two main classes of service discovery approaches: mediator-based discovery approaches and peer-based discovery approaches.

Mediator-based discovery approaches

In mediator-based approaches, service discovery is performed using one or multiple LUS. Services are registered at the LUS. To discover a service, a client requests a desired service at a LUS. Alternatively, the LUS can proactively push service descriptions to clients. Existing mediator-based approaches like Jini [Sun01] or the Intentional Naming System [AWSBL99] cannot provide infrastructure-less operation and do not support sleeping service provider nodes. The Service Location Protocol [GPVD99] allows optional infrastructure-less discovery in case no LUS is found but does not address energy-efficiency.

Peer-based discovery approaches

Direct or peer-based approaches like Universal Plug and Play (UPnP) [MNTW01] or DEAPspace [Nid01] do not rely on the presence of a mediator. In UPnP, to find a service, a client multicasts a discovery request. Nodes offering a suitable service reply with a service description. Alternatively, service providers can proactively multicast descriptions of their services to all potential clients. Communication costs are high due to repeated multicasting – particularly in multi-hop environments. In addition, to receive requests or announcements, nodes must stay awake permanently.

4.4.2 Service Discovery for Resource Poor Devices

To support resource poor devices, a number of research groups (e.g., [HLMW02], [Sun06], [DS01]) has tried to adapt existing technologies, e.g., JINI, to use less resources. Typically, these approaches concentrate on resources like CPU or memory and do not address energy.

JINI Surrogates [Sun06] is Sun Microsystems' solution to use JINI on resource poor end user devices. To do so, each resource poor end user device is assigned to a proxy device that acts on behalf of the end device in the JINI network. This allows to relocate the high resource requirements of JINI to the proxy device. The communication between end device and its proxy can be done using an arbitrary protocol. To allow for more flexibility, the proxy code can be loaded dynamically. The drawback of JINI Surrogates is that it requires stable connections between end devices and their proxies. If the proxy is not available, the end device cannot be discovered or discover any services.

4.4.3 Energy-efficient Service Discovery

The only service discovery system which specifically addresses energy as a first level resource and expilicitely tries to minimize its usage is the service discovery protocol used by IBM Research's DEAPspace project [HHM+00]. DEAPspace provides a service discovery protocol for single-hop MANETs that allows nodes to sleep temporarily [Nid01] [Nid00]. Each node maintains a global view of all services offered in its one-hop environment. It announces this view regularly to all neighbours. If a node receives an announcement that includes all its services, the node omits its own announcement for some time; if one or more of its services are missing, it sends the announcement. Sending announcements is synchronized to a common time window for all nodes, allowing them to sleep in between. An example for the DEAPspace service discovery is given in Figure 4.5.

Here, three nodes A, B, C are present in an environment. A and B already know each other, C has newly arived and is not yet known by the others. At the beginning, each node starts a local timer t_n with a random value from a certain announcement interval. The first node whose timeout occurs, sends its own global view of all services to all nodes. In this example, A sends its global service view. B and C compare this list to their own view of the environment. B detects, that it is already present in A's view, so it must do nothing. C however finds, that it is not yet contained in A's view and thus enters a so-called *panic mode*. In this mode, C uses a considerably shorter random interval to determine its next timeout value t_p. Therefore, C can be sure to be the first, whose timeout occurs and thus sends its own view to the others. To save energy, node B temporairly deactives itself at the beginning of each round. Note, however, that B can only deactive itself long enough to guarantee, that even messages from nodes in panic mode are received.

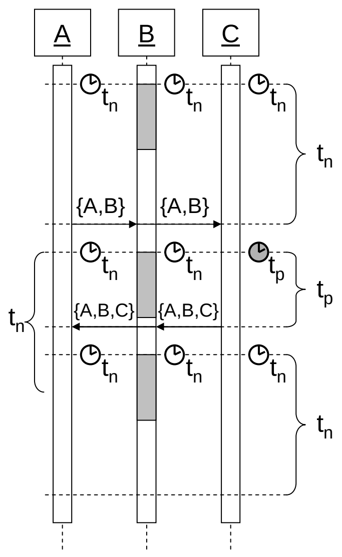


Figure 4.5: DEAPspace service discovery

While DEAPspace provides a simple and efficient way for energy-efficient service discovery in single-hop environments, it has a number of shortcomings. First, it enables maximum node sleep times of approximately 50% of the total node life time [Nid00]. We want to allow much longer sleep periods. Second, it assumes that only a few devices are energy-restricted. With many such devices, the discovery of a new device and its services can be delayed considerably [Nid00]. Third, the global view of services used in DEAPspace is unfeasible for multi-hop environments and therefore less flexible than our approach.

4.4.4 Energy-efficient Communication

Our approach is based on clustering nodes, electing a node to perform service discovery for the cluster and putting the other nodes into sleep mode whenever possible. Related approaches have been proposed for energy-efficient communication, both on the MAC (e.g., [IEE99], [YHE02]) and the routing layer (e.g., [XHE01], [XBM+03], [CJBM02]). Here, the network card is frequently switched between sleep and idle mode to save energy at the cost of a higher communication delay. As an example, the power save mode of IEEE 802.11 switches modes up to 10 times per second. We aim at much higher sleep durations of multiple seconds or more. Approaches that work on the routing layer (e.g., [XHE01], [XBM+03], [CJBM02]) use dynamic node clustering to create a routing backbone. All nodes in the backbone have to stay awake to forward messages, while all other nodes can sleep. If a backbone node receives a message for a sleeping node in its neighborhood, it buffers the message until the node awakes. In contrast to this, our approach allows to answer messages to find a service instantly. Our energy-efficient service discovery builds upon these approaches to provide energy-efficient communication. Using clustering on the application layer allows us to incorporate knowledge about service usage that is not available on the MAC or routing layer.

4.4.5 Summary

Service discovery has been researched in a number of projects and system models. However, existing discovery approaches are either not suitable for resource poor devices or do not address energy efficiency. Therefore, a new approach is needed, which consides both of these requirements. Closely related to our approach, the DEAPspace service discovery allows to deactivate devices temporarily. However, it is only usable in single hop environments and achieves deactivation times of 50% at most. We aim at much higher deactivation times and want to support multi hop communication, too. Another closely related area are energy efficient communication protocols that are based on clustering. However, these protocols induce a large discovery latency, depending on how long devices are deactivated. In contrast to this, we want to achieve discovery latencies that are independent of the deactivation times and are comparable to conventional discovery approaches without deactivating devices.

4.5 Conclusion

In this chapter, we have presented the basic design and architecture of our service discovery system SANDMAN. SANDMAN adapts itself to the stability of different network parts. In highly dynamic parts it keeps nodes active and relies on peer-based discovery to provide low discovery latencies and high discovery precision and recall. In more stable parts SANDMAN dynamically forms node clusters and uses a mediator-based discovery approach in which CHs act as LUS for their clusters. This allows to deactivate idle nodes while preserving low latencies and high precision and recall. SANDMAN is composed of three parts. In the following chapters, we describe these parts in more detail. We start with the cluster management in Chapter 5, present the service management in Chapter 6 and finally the energy management in Chapter 7.

5

Cluster Management

As stated before, the goal of SANDMAN is to deactivate as much idle nodes as possible without loosing the ability to lookup and discover these nodes and their services quickly at runtime. To do so, it partitions the network into a number of disjunct clusters, in which one node acts as cluster head (CH) and is responsible for the discovery of all nodes in its cluster. All other nodes can be deactivated. The election of suitable CNs and the grouping of the remaining nodes into clusters around them is the responsibility of the cluster management sub system. In this chapter, we first derive specific requirements on the SANDMAN cluster management, discuss design decisions and present a suitable protocol. Note, that due to the modular approach taken in SANDMAN, other cluster management protocols may be integrated in the future. After the presentation of our protocol, we discuss related approaches and give a short conclusion and summary of the chapter.

5.1 Requirements

Cluster management is crucial to the energy savings that are achievable with SANDMAN. Choosing 'wrong' nodes as CHs, e.g., transient nodes, may hinder the deactivation of nodes to such an extend, that very little or no energy at all is saved. Therefore, it is important, that the cluster management protocol fulfills a number of specific requirements, which are discussed in the following:

Cluster Stability The cluster management protocol should form stable clusters. This allows SANDMAN to deactivate nodes for long time periods. The reason for this is that cluster changes involving currently deactivated nodes may lead to incomplete or false service discoveries, because the deactivated node is no longer advertised by its former CH and may therefore be temporarily undiscoverable. To restrict this, the duration of the nodes' deactivation times must be adjusted to the achievable cluster stability. If clusters change frequently, nodes must either be activated often or the application must be able to tolerate high discovery inaccuracies.

Fairness As a CH cannot be deactivated in SANDMAN, battery-operated CHs should switch regularly. This allows to distribute the resulting energy consumption for the CHs and results in a longer system lifetime. In addition, energy-rich devices should be preferred to act as CHs over devices with little energy, to further prolong the overall system lifetime.

Cope with Deactivated Nodes The main goal of SANDMAN is to deactivate as many nodes as possible. To not hinder this deactivation, the cluster management has to cope with deactivated nodes at any time. As an example, nodes may not be able to receive *hello* beacons from all their neighbors all the time.

Connectivity Conservation When a node is deactivated, it cannot communicate anymore. When selecting cluster heads, the cluster management must therefore consider the connectivity of the network. Otherwise, deactivating nodes may lead to connectivity losses and even network partitioning. The cluster management must make sure, that it identifies all nodes that are needed to maintain network connectivity and reports them to the energy management such that only nodes that are not needed for connectivity conservation are deactivated.

Clearly, these requirements are partly in conflict with each other, e.g., cluster stability and the dynamic reelection of cluster heads to provide fairness. Therefore, a suitable balance between them has to be found in practice.

5.2 Design Rationale

Our design is aimed at providing a simple and efficient cluster management fulfilling the requirements given before, namely cluster stability, fairness, the ability to cope with deactivated nodes, and connectivity conservation.

In our system model, nodes can leave communication range suddenly and unpredictably. Thus, the cluster management must be able to handle the sudden loss of any node at any time and must still maintain a correct clustering with one CH per cluster and each node being part of at most one cluster. However, nodes can become temporarily unclustered if the situation requires it, e.g., because their CH is lost. The cluster management should try to add as many nodes as possible into clusters but it is free to leave some nodes unclustered if necessary.

Note, that it is neither necessary nor always desired for all nodes in communication range to agree on a single CH and thus form a single cluster. This could lead to rapidly changing clusters, e.g., if a group of highly mobile nodes move past a group of stationary nodes. Due to our requirement of stable clusters, these groups should therefore not be joined into a single cluster and instead should remain two independent clusters. Thus, our design aims to guarantee, that each node is always assigned to at most one cluster and each cluster has always exactly one CH. However, multiple clusters can coexist in the same communication range. To realize this, clusters are formed by first electing a number of nodes to act as CHs and after that assigning other nodes to them. This guarantees that each cluster has only one CH as the cluster is represented by it. If multiple nodes in communication range are elected as CHs this leads to multiple – and thus smaller – clusters. Nodes choose their CH and try to join them. This ensures that each node is always in at most one cluster, as it will never try to join multiple clusters in parallel.

Clearly, this may lead to a larger number of clusters than necessary in some situations. For instance, if two already formed clusters come into communication range and will stay so for a long time, it would be beneficial to join the clusters to save energy, as only one CH would be required. Therefore, we have to provide a means to decide if two clusters should stay independent or should join. In case of a cluster join, the CHs agree on the new - common – CH and hand their clusters over to the new CH. Note, that we do not have to guarantee an agreement, here. If the CHs fail to decide on the new CH, they can stay independent and maintain their clusters, resulting in a less efficient but correct system. In addition, the clustered nodes can always decide to fall back in an unclustered state if the join fails and, e.g., both CHs decide to step back from their role. Although not desirable this ensures that the system is able to operate correctly even in the presence of node and communication failures.

To achieve stable clusters we aim at clustering nodes with similar mobility patterns, i.e., nodes moving together. To avoid connectivity loss, we restrict clustering to nodes that are equivalent in terms of connectivity. Examples for such nodes are devices that are carried by the same user, or devices placed on the same table. In order to provide fairness we elect CHs depending on the node's current battery status. In addition, we rotate the CH role over time. Once CHs are elected, we try to interfere as little as possible with deactivation decisions made by the energy management, allowing nodes to be deactivated at any time. All cluster maintenance operations, e.g., adding a node to the cluster, reassessing a cluster membership, and electing a new cluster head, are designed to work without activating the whole cluster.

5.2.1 Group Mobility

To group nodes with similar mobility patterns, or group mobility, we must first detect such patterns. The most simple approach to do so would be to let the user manually assign nodes to mobility groups, e.g., by connecting devices manually. However, this manual interaction could become quite cumbersome and distractive for users. Therefore, an automated detection technique is needed. Detecting group mobility is relatively easy if we can assume the presence of a location service to obtain node locations. As an example, Wang and Li [WL02] use a location service to acquire movement vectors for all nodes and derive mobility groups by a pair-wise comparison of this vectors. Basu et al. [BKL01] use relative locations, obtained, e.g., by using signal-to-noise ratios, and compare the distance between two nodes over time to estimate their relative mobility. However, the detection quality depends very much on the distance estimation quality which may vary widely due to interferences. In addition, a high micro-mobility, i.e., nodes that move around quickly in a very small area, can lead to multiple groups being discovered.

In our system model we do not require the presence of a location service. Therefore, we use a simple heuristic to detect group mobility. The idea is, that nodes that have stayed in each others vicinity in the past will do so in the near future. Intuitively, nodes that are simply passing by stay in communication range for only a short time. The longer two nodes can communicate, the more likely they have similar mobility patterns. At the very least, waiting a given period of time prevents clustering in highly fluctuating environments. However, the longer the time of observation before clustering, the shorter the time a node can stay in the cluster. Therefore, the time we wait before clustering two nodes must be carefully selected in order to balance group mobility detection and achievable clustered time. We introduce a clustering delay threshold value Δt_{join}, that denotes the minimal duration of two nodes in each others communication range before they can be clustered. By changing Δt_{join}, the cluster management protocol can be adjusted to different intended stability levels at runtime.

5.2.2 Topology Management

To avoid connectivity loss despite deactivated nodes, a suitable topology management has to be included in the clustering protocol. This allows to preserve connectivity by calculating a set of nodes that have to stay active to form a so-called backbone. In general, the problem of finding such a set of nodes can be mapped to the problem of finding a minimum connected dominating set (MCDS). This problem is known to be NP complete even if we assume global knowledge [GJ79]. Therefore, in MANETs typically heuristics are used to approximate a solution. Existing approaches can be classified in three classes: location-based approaches, probability-based approaches and topology-based approaches.

Location-based approaches (e.g., [XHE01], [AKUMK04]) use location information to select backbone nodes. The whole geographic system area is distributed in a number of squares such that each node in a given square can communicate with all nodes in all surrounding squares. After that, one node per square is selected and included in the backbone. As we have discussed earlier, we do not assume the presence of a location service, therefore location-based approaches cannot be used.

Probability-based approaches (e.g., [HCB02], [Xu02]) are independent of any location service. Instead, they use probabilities to elect backbone nodes. As an example, in LEACH [HCB02] each node has a certain chance of entering the backbone depending on a given probability factor and the last time it was a backbone node. However, as decisions are taken independently of other nodes, these approaches can neither guarantee a connected nor minimal backbone.

In contrast to this, topology-based approaches (e.g., [XBM$^+$03], [CJBM02]) use neighborhood information to construct a local view on the current MANET topology. Nodes determine if they should enter the backbone depending on the state of their neighbors.

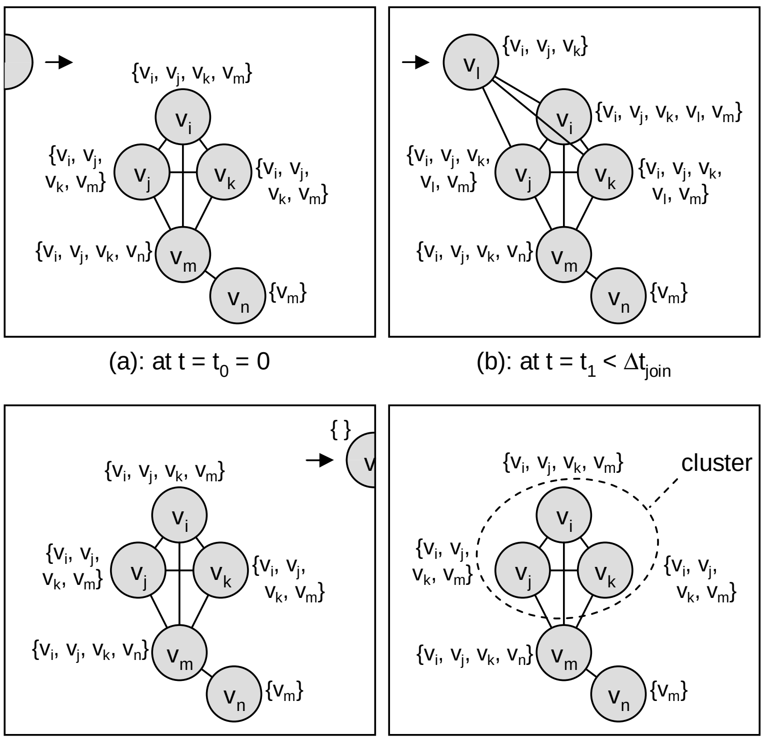
To guarantee connectivity without the need for a location service, we choose a topology-based approach. In addition, we restrict clustering to nodes that are equivalent in terms of connectivity, i.e., nodes that have the same neighbor nodes. This allows to deactivate all CNs in a cluster without connectivity loss and simplifies switching the CH state to another node in the cluster.

5.2.3 Neighborhood Detection

To obtain the topology information necessary to form clusters, each node must detect its current neighborhood. If this information is not provided by a lower layer of the network protocol stack, it must be obtained by the cluster management protocol itself. To do so, we use periodic beacon messages that are broadcasted by nodes periodically to announce their presence. This is a commonly used neighborhood detection technique. To reduce the message overhead, piggybacking of beacons can be used. If a node receives a beacon, it adds the information contained in the beacon to its local neighborhood list. If no beacon is received from a given node for some time, its entry is deleted from the list. In the case of UNs and CHs, the beacon is a special announcement message. For CNs, lease request messages are used to act as beacons, thus reducing the additional message and energy overhead for neighborhood detection. To enable this, lease request messages are broadcasted locally and can therefore be received by all neighboring nodes. Using this neighborhood detection, the nodes are then grouped into clusters.

5.2.4 Cluster Formation and Join

As described before, nodes do not form a cluster immediately if they are found to be in each others communication range. Instead, they compare each others neighborhoods and initiate clustering only if their neighborhoods are the same for at least the clustering delay Δt_{join}. This is typically the case for nodes that are located near to each other and move together as a mobility group. The most prominent example for such a group are nodes that are carried by the same user. Nodes that pass by the group are not clustered with it, as they do not stay in communication range long enough. This allows to filter a group of nodes from the whole network, despite other nodes being present. An example for this process is given in Figure 5.1. At the start (see Figure 5.1 (a)), there are five nodes in the system. Nodes v_i, v_j, v_k and v_m are all in communication range with each other. However, only v_m can communicate with v_n. This is shown in the neighborhood information provided for each node. After some time (see Figure 5.1 (b)), a new node v_l moves in the vicinity and gets in communication range with v_i, v_j and v_k. However, it does not stay long enough to be clustered and leaves at time $t_2 < \Delta t_{join}$ (see Figure 5.1 (c)). All other nodes stay and after Δt_{join}, v_i, v_j and v_k are grouped into a cluster (see Figure 5.1 (d)). Although v_m is also in communication range with the other nodes for the whole time, it is not included in the cluster, as its neighborhood differs to that of v_i, v_j and v_k.



(a): at t = t_0 = 0

(b): at t = $t_1 < \Delta t_{join}$

(c): at t = $t_2 < \Delta t_{join}$

(d): at t = $t_3 \geq \Delta t_{join}$

Figure 5.1: Cluster Formation Example

While this approach does not guarantee that only nodes that have the same mobility patterns are grouped, using Δt_{join} the cluster management can be parameterized to make the clustering of nodes with different mobility patterns increasingly improbable. However, with increasingly large values for Δt_{join}, nodes may not be clustered at all. This is a tradeoff between cluster stability and the probability of nodes being clustered.

As described before, clusters are formed by first electing a CH and letting the other nodes join its new cluster afterwards. This guarantees that each cluster has always exactly one CH. Therefore, the initial cluster formation is a special case of CH election. If two unclustered nodes detect that they had the same neighborhood for at least Δt_{join}, they initiate the CH election process as discussed in the next section. One of them is elected as the new CH and the other joins the new cluster.

Another case for joining a cluster is if two or more CHs with existing clusters come into communication range. Again, the CHs do not join immediately to make sure that the clusters are not just co-located temporarily. Instead, the CHs use the exact same mechanism to delay their join as unclustered nodes. They compare their neighborhoods and wait for Δt_{join} before initiating a cluster join. After this time, the system assumes that the two clusters have the same mobility patterns. Therefore, it is beneficial to join the clusters as this allows to use one CH instead of two, resulting in higher energy savings. The election of the new CH is described in the next section. How the clusters are actually joined after electing a new CH is shown in Section 5.2.8.

5.2.5 Cluster Head Election

A CH is elected in three different situations. First, if a new cluster should be formed from previously unclustered nodes. Second, if two or more clusters have decided to join each other into one new cluster. Third, if a CH wants to give up its role in order to achieve the fairness requirement discussed before. All these situations must be reflected in the CH election process.

In general, if multiple nodes compete for the CH role, we have to select one of them. To do so, typical cluster management protocols introduce some metric to compare the suitabilities of two nodes to act as CH. We call this the *CH suitability metric* or *CH suitability* for short. The choice of which properties to use as CH suitability is an important design decision. By selecting different properties, the cluster management protocol can be aimed towards different design goals. As an example, [BKL01] uses the node's relative mobility to select the node which moves the least and therefore select stable CHs. In general we can distinguish between static and dynamic CH suitabilities. Static suitabilities are fixed, i.e., a node that has a higher suitability at any time in the past will do so at any time in the future. The most prominent example for a static suitability is lowest ID [EWB87], i.e., the node with the lowest identifier becomes CH. Dynamic suitabilities can change over time, i.e., the ordering between some nodes' suitabilities may change. One well-known example for this is max-connectivity [GT95]. Here, the node with the highest number of neighbors becomes CH.

In SANDMAN, we aim at maximizing the lifetime of all nodes. This is because we assume that all nodes are equally important and should therefore be held available as long as possible. In addition, we assume cooperative users that are willing to invest energy to keep other user's nodes alive. Therefore, similar to [XHE01] [XBM$^+$03] we use the remaining energy of a node as its primary suitability metric. A node with more energy is better suited to act as CH as a node with little energy left. This allows energy-poor devices to save energy while a energy-rich device acts as CH. We chose to use the remaining lifetime instead of, e.g., the remaining energy, to support heterogeneous devices with different power consumptions. In case the remaining lifetimes are equal, the unique node identifier is used as a tie breaker.

In addition, CH suitabilities are used to reelect the CH role in a cluster. As a CH cannot be deactivated in SANDMAN and therefore consumes more energy than a CN, each CH will eventually have less energy left than its CNs. Therefore, its remaining lifetime will become less than that of one or all of its CNs and the CH role will switch to another CN. However, without further precautions the new CHs lifetime would drop below that of its new CNs fastly, leading to oscillating role switches. Therefore, we introduce a threshold value γ. When comparing two lifetimes, they must differ by more than γ to be considered different. Otherwise, they are assumed to be equal. This allows to define the frequency of switching the CH role in a cluster by choosing different values for γ. We chose to use a static threshold value, e.g., 5 minutes, instead of a dynamically adjusted one, e.g., 5% of the lifetimes, to keep the CH reelection rate independent of the remaining lifetimes. Otherwise, if the lifetimes become shorter, the system might switch between CHs increasingly fast, consuming more and more energy and further shortening the lifetimes.

Note, that for γ to work properly, we have to omit using the node identifiers as a tie breaker when comparing the suitabilities of a CH and a CN. To explain this, we give a short example. We assume a cluster containing two nodes v_1 and v_2, with v_1 being the current CH and continuously comparing the nodes' suitabilities. As the CH, v_1 consumes more energy. Therefore, its remaining lifetime rl_{v_1} declines faster than that of v_2. At one point, v_1 detects, that $|rl_{v_1} - rl_{v_2}| < \gamma$. Therefore, the node IDs id_{v_1} and id_{v_2} are compared and v_2 wins, leading to v_1 handing its cluster over to v_2. Now, v_2 is the CH and consumes more energy than v_1. Thus, the next time the suitabilities are checked again, the system detects that rl_{v_1} is again larger than rl_{v_2} even considering γ and the cluster should be handed back to v_1. This behavior is independent of the size of γ and must be avoided. To do so, we additionally include the current roles of the nodes into the suitability comparison to make sure that when comparing a CH with one of its CNs, the current CH always wins as long as the lifetimes differ by less than γ regardless of the node IDs. The resulting CH suitability is given by a suitability triple as defined in Definition 5.1.

Definition 5.1 (suitability triple) *A suitability triple $s_{v_i} := (id_{v_i}, r_{v_i}, rl_{v_i})$ of a node v_i consists of the node's unique identifier id_{v_i}, its current role r_{v_i}, and its remaining battery-lifetime rl_{v_i}.*

To compare two suitability triples, we define a relation $<_s$ between two suitability triples $s_{v_i} = (id_{v_i}, r_{v_i}, rl_{v_i})$ and $s_{v_j} = (id_{v_j}, r_{v_j}, rl_{v_j})$ as:

$$\begin{aligned} <_s := \{(s_{v_i}, s_{v_j}) \mid \; & (rl_{v_i} + \gamma < rl_{v_j}) \vee (|rl_{v_i} - rl_{v_j}| \leq \gamma \wedge r_{v_i} = CN \wedge r_{v_j} = CH) \\ & \vee (|rl_{v_i} - rl_{v_j}| \leq \gamma \wedge (r_{v_i} \neq CH \vee r_{v_j} \neq CN) \wedge id_{v_i} < id_{v_j})\} \end{aligned} \tag{5.1}$$

Informally, two suitabilities are compared by first comparing their lifetimes. If the lifetimes differ by more than a given threshold value γ, the suitability with the higher lifetime wins. Otherwise, if the first node is a CN and the second is a CH, the second wins. In all other cases, the identifiers are compared and the suitability with the higher identifier wins.

To perform a CH election, the participants have to know the CH suitabilities of the potential CHs. To ensure this, we include the CH suitability in the beacons send for the neighborhood discovery (see Section 5.2.3). Upon receiving a beacon, the receiver checks if it could form a cluster with the sender. If so, the receiver compares its own suitability with that included in the beacon, using the relation v_s. If the sender's suitability is higher than the receiver's, the receiver decides to join the sender. This algorithm is described in more detail in Section 5.4. To join the cluster, the joining node contacts its new CH and asks to be included in its cluster. See Section 5.5 for more information about this algorithm.

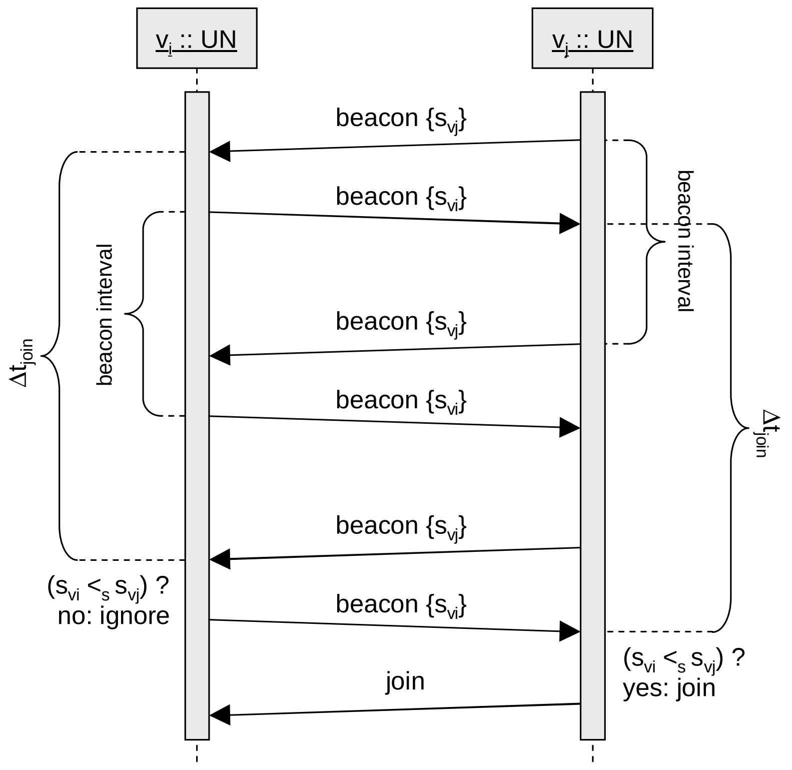


Figure 5.2: Cluster Head Election Example

A short example is given in Figure 5.2. Here, nodes v_i and v_j send each other beacon messages including their own CH suitability regularly. After Δt_{join}, v_i detects that it can join with v_j. However, it finds, that its own suitability is larger than v_j's. Therefore, it does not initiate a join. The node v_j on the other hand, detects that its suitability is lower than v_i's. Thus, it initiates a join by sending a corresponding message to v_i. Note, that only one node will try to join the other, as $<_s$ results in a total order of all CH suitabilities.

5.2.6 Cluster Membership

Once clustered, nodes use a well-known soft-state approach to maintain cluster membership. To stay in the cluster, each CN regularly sends a lease renewal message to its CH. The CH acknowledges this message. This way, both CN and CH know that they are still in communication range. If a CH does not receive a lease renewal message from a CN in time, it will remove this CN from its cluster. Likewise, if the CN does not receive an acknowledgment for its lease renewal, it leaves the cluster, switches back to the UN state and starts over.

5.2.7 Periodic Probing

Before adding a node into a cluster, the cluster management checks, that the new node has the same neighborhood as the current CH, as described in Section 5.2.4. However, after the node has joined the cluster and became a CN, this condition is not checked anymore. Instead, as long as the CN is in communication range with its CH, it stays in the cluster. This can cause problems, if the connectivity in the CN's and the CH's neighborhoods change in such a way, that their connectivities start to differ from each other. To explain, why connectivity changes for CNs should be detected, consider the following example, shown in Figure 5.3. At a given time, a system consists of two nodes v_i and v_j (see Figure 5.3 (a)). Both are located far away from each other but are still in communication range. As there are no further neighbors, the clustering algorithm will cluster them eventually (see Figure 5.3 (b)). At a later time, a third node v_k enters the system which is in communication range with v_i but not with v_j (see Figure 5.3 (c)). If in this situation v_j is the CH and v_i is deactivated, a network partition occurs until the CH role is forwarded to v_i. A second problem may arise if the nodes are clustered correctly but move afterwards. As an example, assume that in the system described above, v_k moves towards v_i (see Figure 5.3 (d)), until it comes in communication range with v_j (see Figure 5.3 (e)). The clustering algorithms will eventually join all nodes into a single cluster (see Figure 5.3 (f)). Now, while v_j is the CH, v_k moves further, leaving v_i's communication range (see Figure 5.3 (g)). At some time, v_j hands the CH role over to v_k. In this situation v_i has lost contact to the CH and will leave the cluster (see Figure 5.3 (h)). We can create increasingly severe situations of this kind by adding more nodes at v_i's location. In addition, if v_k moves further away, it will eventually leave the communication range of v_j, too. This results in the cluster being dissolved

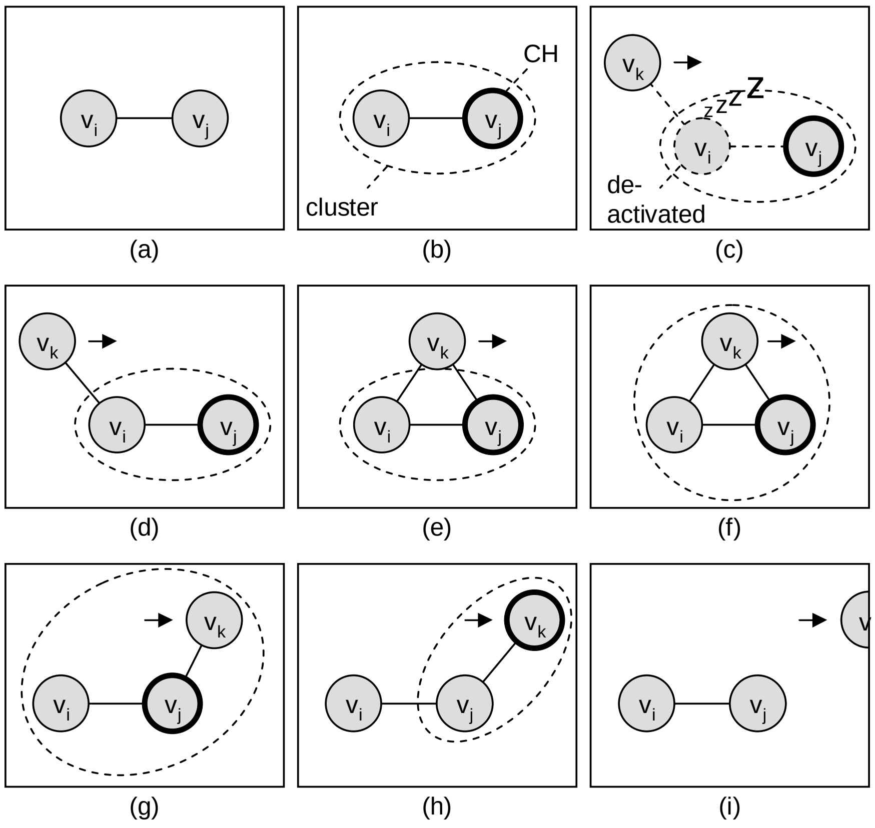


Figure 5.3: Periodic probing scenario

(see Figure 5.3 (i)). If v_j would have detected this situation before handing its CH role to v_k, this could have been avoided and the cluster could have been preserved.

To detect connectivity changes for CNs, we use so-called *periodic fully awake intervals* as proposed, e.g., in [THH02]. We call this mechanism *periodic probing*. Periodically, the CH instructs each CN to enter the probing sub state. In this state, a CN stays active for a complete cycle and tries to detect all its neighbors. After probing is finished, the CN returns its updated neighborhood list to the CH, which compares it with its own list. If the lists differ, the original clustering condition – the CN and its CH sharing the same neighborhood – is no longer valid and the two nodes should not be clustered anymore, in order to avoid the problems discussed before. Therefore, the CN is removed from the cluster and returns to the UN state. To implement this simple mechanism, each node must be detectable for a probing CN during a predefined period of time. Thus, a system-wide fixed maximum lease duration is introduced. The probing phase length can then be set to the maximum of the announcement message frequency and the maximum lease length.

5.2.8 Cluster Handover

As discussed in Section 5.2.4, clusters are joined if their CHs have been found to be clusterable. This allows to lower the number of simultaneously active CHs in the system and thus to save additional energy. To do so, the CHs first agree, which one of them should be the new CH (see Section 5.2.5). The others join the new CH's cluster. Once a CH has decided to join another cluster, it must decide what should happen with its current cluster. The most straightforward approach is to simply abandon the cluster. In time, all CNs become unclustered and restart searching for a CH. However, due to the clustering delay Δt_{join}, this leads to unnecessary activation times and communication. Therefore, when joining the new cluster, the CH transfers all data about its old cluster to the new CH. After this transfer is complete, the old CH can be deactivated instantly, saving additional energy. If a CN tries to renew its lease, the new CH intercepts this message and sends a lease extension by itself. No additional activation time for CNs is needed. To enable the new CH to intercept lease renewals, they are sent using local broadcast. An alternative approach without the need to intercept lease renewals would be for the old CH to stay active for an additional round of lease renewals and to relay each CN to the new CH consecutively. This approach would be beneficial if the additional overhead of overhearing lease renewal requests outweighs the CH staying active for a longer period of time. In our implementation, lease renewals are broadcasted anyway to enable neighborhood detection, therefore causing no additional overhead for cluster handovers.

An example for our approach for cluster handovers is given in Figure 5.4. There are three nodes v_i, v_j and v_k. At the beginning, v_j is the CH, while v_i and v_k are its CNs. To extend its lease, v_i

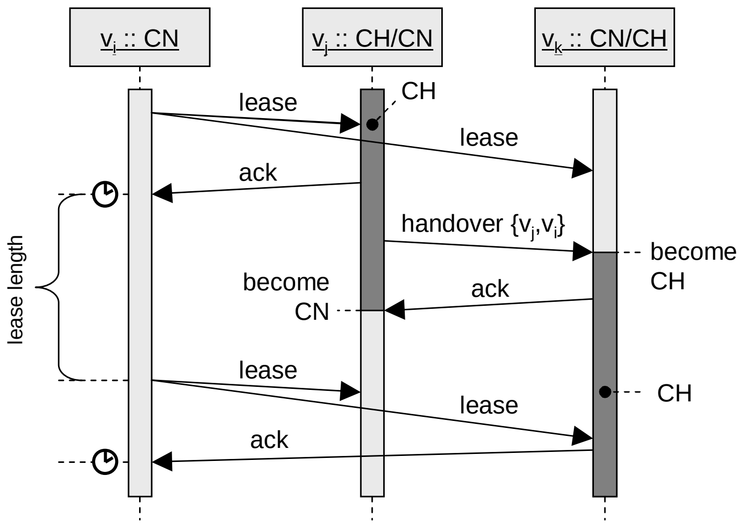


Figure 5.4: Handover scenario

broadcasts a lease renewal message in its one hop neighborhood. Upon receiving the message,

v_j finds, that it is v_i's CH and thus returns an acknowledgement message to it. Later, v_j decides to hand its cluster over to v_k. To do so, it sends a handover message – containing the nodes in the cluster – to v_k, which acknowledges the handover and takes over as CH. The next time v_i sends its lease renewal message, v_j is no longer responsible and thus ignores the message. Instead, the new CH v_k answers the message, notifying v_i of the cluster handover.

5.3 Data Structures and Variables

Having discussed the design rationale of our approach and giving an overview of it, we now describe the cluster management algorithms in more detail. To do so, we first introduce a number of data structures and variables that we use in the description. After that, we present the algorithms in Section 5.4 to 5.6.

5.3.1 Data Structures

The cluster management relies on two data structures, *NeighborSet* and *Cluster*, which are presented in the following.

NeighborSet

Each node manages a local NeighborSet. This set includes the unique identifiers of all known nodes in single hop distance and a lifetime for each entry. If the lifetime expires, the entry is removed from the neighborhood automatically. This ensures that the NeighborSet is kept up to date. NeighborSet offers two operations: `Add` and `GetAll`.

Add(SystemID, lifetime) The `Add` operation is called to enter a new neighbor into the NeighborSet or to update an existing entry's lifetime. The operation takes two parameters, a unique identifier and a lifetime value.

GetAll() returns SystemID[] To get a identifier list of the current neighborhood, the `GetAll` operation is called. It takes no parameters and returns an array of SystemIDs.

Cluster

A *Cluster* contains information about all CNs in a cluster. For each CN it includes the CN's SystemID *id*, the time of the last probing phase *probe*, and the remaining lease length *lease*. In addition, Cluster contains a timer to detect if the lease of any of the CN expires. If this happens, Cluster fires a *LeaseExpired* event (see Section 5.6.1). Cluster offers six operations, which are described in the following. In addition, it offers operations to subscribe and unsubscribe for LeaseExpired events. These operations are omitted here for simplicity.

Add(SystemID, leaseLength) To add a new CN to its cluster, a CH can call the `Add` operation with the CN's SystemID and a lease length. This includes the CN in the Cluster and starts the timer for the given lease length. The time of the CN's last probing phase is set to the current time t_{now}.

Renew(SystemID, leaseLength) To renew the lease of a CN, a CH calls the `Renew` operation. `Renew` takes the CN's SystemID and the new lease length as its parameters. It searches for the specified CN in the Cluster and changes its lease length accordingly.

GetLastProbing(SystemID) returns time stamp The time at which a CN ended its last probing phase can be queried by calling the `GetLastProbing` operation with the CN's SystemID. This returns the time stamp of the last probing.

Remove(SystemID) The `Remove` operation removes a CN from the cluster. It takes the SystemID of the CN to remove as its single parameter.

Merge(Cluster) In case of a hand over, two clusters may need to be merged. To do so, the `Merge` operation can be used. The caller provides the Cluster to merge as parameter to the operation. When `Merge` returns, all entries in the old cluster are included in the new Cluster.

Contains(SystemID) returns Boolean Lastly, the `Contains` operation checks, if a CN, given by its SystemID, is a member of this Cluster. If so, it returns `true`, `false` otherwise.

5.3.2 Variables

In our protocol specification, we use a number of variables that are described in Table 5.1. Most of them are used to access local state information of a node. As an example, *id* denotes the local node's unique system identifier. The current system time can be accessed using the t_{now} variable.

variable	**description**
id	the unique SystemID of the local node
$life$	the remaining lifetime of the local node
$neighbrs$	a local instance of NeighborSet
$neighbrs(t)$	a copy of *neighbrs* made at time t
$role$	the current role of the local node
ch	the unique identifier of the local node's current CH, if any
$suit$	the CH suitability of the local node
$cluster$	a local instance of Cluster
$joining$	`true` if the local node is currently joining
$handover$	`true` if the local node is currently handing over its cluster
$probing$	`true` if the local node is in a probing phase
$probeEnd$	the time at which a running probing will end
$lease$	the time at which the node's lease will expire
t_{now}	the current system time

Table 5.1: Variables used by the cluster management

5.4 Algorithms: Neighborhood Detection and Cluster Preparation

At system startup time, all nodes are unclustered, i.e., they are in the UN role. In this state, all nodes periodically send ANC messages to announce their presence to other nodes and try to detect their current neighborhood in order to find nodes that could be used to form clusters.

5.4.1 Announcements

Announcements are sent periodically by nodes in the UN or CH role. An ANC message includes the sender's current role, its remaining lifetime and its currently known neighborhood. As described in Section 4.3.3, the sender's unique identifier can be read by accessing the variable *sender* at reception time. The identifier, role, and remaining lifetime are used to determine the sender's CH suitability. Its neighborhood is used to determine if the recipient and the sender could form a cluster together.

AnnouncementTimeout

To implement the periodic broadcasting of ANC messages by UNs and CHs, each node maintains a *AnnouncementTimeout* timer. When an *AnnouncementTimeout* occurs, the `OnAncTimeout` operation (see Procedure 5.1) is executed. The node first checks if it is currently a UN or CH. If so, it sends an ANC message including its remaining lifetime, its current role, and and its current neighborhood. Finally, it restarts the *AnnouncementTimeout* timer with $\frac{1}{f_{anc}}$.

Procedure 5.1 *OnAncTimeout*: Handling an AnnouncementTimeout event

```
OnAncTimeout()
1: if $role$ = (UN $\vee$ CH) then
2:   bsend(ANC,$\{role, life, neighbrs\}$);
3:   startTimer(AnnouncementTimeout, $\frac{1}{f_{anc}}$);
4: end if
```

Announcement Messages

On reception of an ANC message, the `OnAnc` operation is executed (see Procedure 5.2). In this operation, the receiver first updates its neighbor list by calling the corresponding `Add` operation.

Procedure 5.2 *OnAnc*: Receiving an ANC message

```
OnAnc( SystemID $sender$, Role $r$, long $l$, NeighborSet $ns$ )
1: $neighbrs$.Add($sender$, $t_{anc}$);
2: if $role$ = UN $\wedge \neg joining$ then
3:   if $suit <_s \{sender, r, l\}$ $\wedge$ isJoinable($sender$, $ns$) then
4:     $joining$ := true;
5:     usend($\{sender$, ClusterManager$\}$, JOIN_REQ, $\{\}$);
6:     startTimer(JoinRequestTimeout, $to_j$);
7:   end if
8: else if $role$ = CH $\wedge \neg handover$ then
9:   if $suit <_s \{sender, r, l\}$ $\wedge$ isJoinable($sender$, $ns$) then
10:    $handover$ := true;
11:    usend($\{sender$, ClusterManager$\}$, HO_REQ, $\{$prepHandover($clust$)$\}$);
12:    startTimer(HandoverRequestTimeout, $to_h$);
13:  end if
14: end if
```

In addition, if the node is currently in the UN role and not already joining another cluster, it checks if it should join the sender's cluster. If so, the node enters the joining state, sends a JOIN_REQ message to the sender and starts a *JoinRequestTimeout* with a timeout value to_j. Similarly, if the node is in the CH role and not handing over its cluster already, it checks if it should join the sender's cluster. If so, the node enters the handover state, sends a HO_REQ message to the sender including its cluster, and starts a *HandoverRequestTimeout* using a timeout value of to_h. Note, that the CH transfers not the original cluster data but a modified version obtained from the `PrepHandover` procedure. This procedure adds a minimal lease length $lease_{min}$ to all CN leases in the cluster to ensure that lease renewals work during handover. See Section 5.6.1 for more details on how this works.

5.4.2 Detecting Joinable Nodes

To determine if a given node could be joined, Procedure 5.2 uses the `IsJoinable` operation, given in Procedure 5.3. It first checks, if the node has been a neighbor for a least Δt_{join}. If so, it compares the node's neighborhood. If it is equal to the local neighborhood, the node can be joined. In all other cases the node cannot be joined.

Procedure 5.3 *IsJoinable*: Determine if a given node could be joined

```
IsJoinable(SystemID i, NeighborSet ns) returns boolean
1: if $(ns = neighbors) \wedge (\forall t \in [t_{now} - \Delta t_{join}, t_{now}] : i \in neighbors(t))$ then
2:     return true;
3: else
4:     return false;
5: end if
```

5.5 Algorithms: Cluster Creation and Join

To join a cluster, a UN sends a JOIN_REQ message to the node it wants to join. On reception of the message, the `OnJoinReq` operation (see Procedure 5.4) is called.

Procedure 5.4 *OnJoinReq*: Receiving a JOIN_REQ message

```
OnJoinReq(SystemID sender)
1: if $(role = \text{UN} \wedge \neg joining) \vee (role = \text{CH} \wedge \neg handover)$ then
2:     if $role$ = UN then
3:         SwitchRole(CH);
4:     end if
5:     $cluster$.Add($sender$, $lease_{init}$);
6:     usend({$sender$, ClusterManager}, JOIN_ACK, {$lease_{init}$});
7: else
8:     usend({$sender$, ClusterManager}, JOIN_NACK, {});
9: end if
```

If the receiver is in the UN role and currently not joining another cluster, it becomes a CH and includes the sender in its newly formed cluster. Similarly, if the receiver is in the CH role and currently not handing over its cluster, it adds the sender to its cluster. In both cases, the sender is granted a short initial lease $lease_{init}$. The initial lease is intended to give the new CN enough time to request a full lease by sending a lease request message. This allows to send JOIN_REQ messages as unicasts while preserving the ability of probing CNs to detect newly joined CNs, as the latter will request a new lease soon after joining. To inform the sender of the successful cluster addition, the receiver returns a JOIN_ACK acknowledgement message to the sender. In all other cases, the receiver denies the join by sending a JOIN_NACK negative acknowledgement message to the original sender.

Switch Node Role

To consistently switch between node roles, the `SwitchRole` operation shown in Procedure 5.5 is used. It sets the new role and initializes the *joining*, *handover* and *probing* variables accordingly. In addition, if the node switches to the roles UN or CH, it immediately sends an ANC message and starts the *AnnouncementTimeout* timer to be notified after $\frac{1}{f_{anc}}$ seconds. If the node becomes a CN, it stops the timer to stop sending announcements.

Procedure 5.5 *SwitchRole*: Switch node role

```
SwitchRole(Role $r$)
1:  $role := r$;
2:  **if** $r$ = UN **then**
3:      $joining$ := false;
4:      bsend(ANC,$\{role, life, neighbrs\}$);
5:      startTimer(AnnouncementTimeout, $\frac{1}{f_{anc}}$);
6:  **else if** $r$ = CH **then**
7:      $handover$ := false;
8:      bsend(ANC,$\{role, life, neighbrs\}$);
9:      startTimer(AnnouncementTimeout, $\frac{1}{f_{anc}}$);
10: **else if** $r$ = CN **then**
11:     $probing$ := false;
12:     stopTimer(AnnouncementTimeout);
13: **end if**
```

JOIN_ACK Message

After successfully adding a node into its cluster, a CH sends a JOIN_ACK message to the node (see Procedure 5.6). The receiver first checks, if it is still waiting for an answer to its JOIN_REQ message. If not, the message is omitted. Otherwise, the receiver stops the *JoinRequestTimeout* timer, switches to CN role and sets the sender of the JOIN_ACK message as its new CH. Finally, it starts the lease mechanism by starting a *LeaseRenewalTimeout* timer with the initial lease length given in the JOIN_ACK message.

Procedure 5.6 *OnJoinAck*: Receiving a JOIN_ACK message

OnJoinAck(SystemID *sender*, long *lease*)

```
1: if $role$ = UN $\wedge$ $joining$ then
2:     stopTimer(JoinRequestTimeout);
3:     $ch$ := $sender$;
4:     switchToRole(CN);
5:     startTimer(LeaseRenewalTimeout, $lease$);
6: end if
```

JOIN_NACK Message

A JOIN_NACK message is send as the result of a declined JOIN_REQ message. On reception, the `OnJoinNAck` procedure (see Procedure 5.7) is called. If the receiver is currently in the process of joining, the joining is canceled and normal operation is resumed. Otherwise, the JOIN_NACK message is omitted.

Procedure 5.7 *OnJoinNAck*: Receiving a JOIN_NACK message

OnJoinNAck(SystemID sender)

```
1: if $role$ = UN $\wedge$ $joining$ then
2:     stopTimer(JoinRequestTimeout);
3:     $joining$ := false;
4: end if
```

Join Request Timeout

A *JoinRequestTimeout* timer is started by a UN when it sends a JOIN_REQ message to its potential new CH. If no answer is received in time, a *JoinRequestTimeout* occurs. This timeout is handled exactly like the reception of a JOIN_NACK (see Procedure 5.7).

5.6 Algorithms: Cluster Maintenance

Once a node has successfully joined a cluster, it sends LEASE_REQ messages periodically to renew its lease at the CH and to stay in the cluster. To do so, it operates a *LeaseRenewalTimeout* timer, which notifies it each time that its lease must be renewed.

5.6.1 Lease Renewal

Each time a *LeaseRenewalTimeout* occurs (see Procedure 5.8), the CN tries to extend its lease at the CH. To do so, the CN sends a LEASE_REQ message to its CH containing the new lease length an it remaining lifetime. In addition, the CN checks if is currently in a probing phase and if this probing could be finished, because the probing duration has expired. If so, the CN leaves the probing phase and adds its updated NeighborSet to the LEASE_REQ message.

To detect a lost CH, the CN starts a local timer that issues a timeout if no answer from the CH has been received in time. For the sake of simplicity, we abstract from this asynchronous mechanism in our pseudo code. Instead we introduce a variable $successful$ that can be accessed to block until the transmission has been successful or the timeout has occured. If contacting the CH was unsuccessful, the operation leaves the cluster and switches back to UN role. Otherwise, the CH has send us a message and system execution continues with another operation handling this message.

Procedure 5.8 *OnLeaseRenewalTimeout*: Initiating a lease renewal

```
OnLeaseRenewalTimeout()
1: if $role$ = CN then
2:   if ($probing$ = true) $\wedge$ ($t_{now} \geq probeEnd$) then
3:     $probing$ := false;
4:     bsend(LEASE_REQ,$\{ch, lease, life, neighbrs\}$);
5:   else
6:     bsend(LEASE_REQ,$\{ch, lease, life,$ null$\}$);
7:   end if
8:   if not successful then
9:     SwitchRole(UN);
10:  end if
11: end if
```

LEASE_REQ Message

A LEASE_REQ message contains the intended length of the new lease and the requesting CN's NeighborSet if this LEASE_REQ message is the first after the end of a probing phase. Otherwise it contains `null` instead of the NeighborSet. LEASE_REQ messages are broadcast to all nodes in single hop distance, to allow others to detect the CN.

Upon reception of a LEASE_REQ message, a node executes the `OnLeaseReq` operation shown in Procedure 5.9. The receiving node first updates its local neighborhood. After that, it checks if it is the sender's CH and omits the message if not. Now, the receiver determines if the message contains an updated NeighborSet and if so, compares it with its own. If both are equal, the CH knows that both nodes share the same connectivity and can therefore stay in one cluster. Otherwise, the CN should be removed and the CH omits the message without granting a new lease. If the CN can stay in the cluster, the CH determines if it should hand its cluster over to this CN. In this case, a HO_REQ message is send. Otherwise the CH checks if the CN should enter its periodic probing phase using the probing frequency f_{prob} and calculates the length of the granted lease. If the CH is currently trying to handover its cluster, it grants only a minimal lease $lease_{min}$, otherwise the CH grantes the lease length requested in the LEASE_REQ message. Finally, a GRANT_LEASE message containing the new lease length and a boolean indicating

Procedure 5.9 *OnLeaseReq*: Receiving a LEASE_REQ message

```
OnLeaseReq(SystemID $sender$, long $le$, long $li$, NeighborSet $ns$)
1: $neighbrs$.Add($sender$, $l$);
2: if $role$ = CH $\wedge$ $sender \in cluster$ then
3:   if $(ns \neq \emptyset) \wedge (ns \neq neighbrs)$ then
4:     abort;
5:   end if
6:   if $suit <_s \{sender$,CN,$li\}$ then
7:     $handover$ := true;
8:     usend({$sender$, ClusterManager}, HO_REQ, {prepHandover($cluster$)});
9:     startTimer(HandoverRequestTimeout, $to_h$);
10:  else
11:    boolean $p := (t_{now} >$ cluster.GetLastProbing($sender$) + $1/f_{prob}$);
12:    if $handover$ then
13:      $le := lease_{min}$;
14:    end if
15:    $cluster$.Renew($sender$,$le$);
16:    usend({$sender$, ClusterManager},GRANT_LEASE, {$le$, $p$});
17:  end if
18: end if
```

whether probing should be started or not is send to the CN to renew the lease. As explained before, while a CH performs a cluster handover, it grants only minimal leases to its CNs. The reason for this is, that the CH has already transferred its cluster data to the new CH. The new CH does not know, that the CN renewed its lease at the old CH and may remove it by mistake. By granting a minimal lease, the old CH effectively orders its CN to retry its lease renewal soon, hopefully after the new CH has taken control. The new CH can then grant a full lease again. The minimal lease renewal can be tolerated by the new CH, because the old CH adds it to all leases before transferring the cluster, as shown in Section 5.4.1.

GRANT_LEASE Message

If a CH accepts a LEASE_REQ, it sends a GRANT_LEASE message to the CN containing the length of the granted lease and a boolean specifying if the receiving CN should enter its probing phase.

Procedure 5.10 *OnGrantLease*: Receiving a GRANT_LEASE message

```
OnGrantLease(SystemID $sender$, long $l$, boolean $p$)
 1: if $role$ = CN then
 2:     if $l > lease$ then
 3:         startTimer(LeaseRenewalTimeout, $l$);
 4:         if $p$ = true then
 5:             $probing$ := true;
 6:             $probeEnd := t_{now} + \Delta t_{probe}$;
 7:         end if
 8:         if $ch \neq sender$ then
 9:             $ch := sender$;
10:         end if
11:     end if
12: end if
```

Using the `OnGrantLease` operation shown in Procedure 5.10 the receiver first checks, if it is a CN and compares the granted lease length to its currently existing lease. Only leases, that prolong the existing lease are accepted. This is, because during handover there is a small times-pan, during which it is possible, that a CN's LEASE_REQ message is processed and answered by both the old and the new CH. Depending on the order in which the answers are received, the old CH's GRANT_LEASE message could overwrite the new CH's. However, the old CH grants only a short minimal lease $lease_{min}$ while the new CH grants a full lease. Therefore, by checking the lease length on reception, we can guarantee, that the new CH always wins. After checking the length of the lease, the receiver restarts its internal *LeaseRenewalTimeout* timer with the new lease length and starts probing for the duration of a probing interval Δt_{probe} if needed. Finally, it checks if the sender of the GRANT_LEASE message is its own CH. If not, this answer is the first from the new CH after a cluster handover and the receiver updates its local *ch* variable.

Lease Expired

A LeaseExpired event is created by the Cluster container if an entry in the cluster has expired, i.e., if the lease for a CN contained in the cluster has not been renewed in time. The event is handled by the `OnLeaseExpired` operation (see Procedure 5.11).

Procedure 5.11 *OnLeaseExpired*: Handling a LeaseExpired event

```
OnLeaseExpired(SystemID $i$)
1: $cluster$.Remove($i$);
2: if ($role$ = CH) $\wedge$ ($\neg handover$) $\wedge$ ($cluster = \emptyset$) then
3:   SwitchRole(UN);
4: end if
```

First, the CN is removed from the cluster. If the cluster is now empty, the node switches to the UN role as a CH is defined to have a non-empty cluster. However, this is only true, if the node is not currently in the process of handing over its cluster to another CH. While a handover is running, a node does not revert to UN state, to not compromise the handover.

5.6.2 Cluster Handover

If a CH decides to merge its cluster with another one, it sends a HO_REQ message to the new CH, including its node cluster *cluster*.

HO_REQ Message

When a HO_REQ message is received, the `OnHandoverReq` operation is called (see Procedure 5.12). This operation resembles the *OnJoinReq* procedure in large parts.

Again, a UN that is not joining and a CH that is not performing a handover add the sender to their cluster and return a HO_ACK message. In addition, both add the sender's cluster to their own cluster. Furthermore, a CN accepts a handover to allow a CH to rotate their role in their cluster. It switches to CH role and initializes its new cluster with the previous CH and its node cluster. In all other cases, the receiver denies the handover and sends a HO_NACK message to the initial sender.

Procedure 5.12 *OnHandoverReq*: Receiving a HO_REQ message

```
OnHandoverReq(SystemID $sender$, Cluster $c$)
1: if ($role$ = UN $\wedge \neg joining$) $\vee$ ($role$ = CH $\wedge \neg handover$) $\vee$ ($role$ = CN) then
2:     if $role \neq$CH then
3:         SwitchRole(CH);
4:     end if
5:     $cluster$.Merge($c$);
6:     $cluster$.Add($sender$,$lease_{init}$);
7:     usend({$sender$,ClusterManager}, HO_ACK,{$lease_{init}$});
8: else
9:     usend({$sender$,ClusterManager}, HO_NACK,{ });
10: end if
```

HO_ACK Message

A HO_ACK message is send by a CH to its new CN after receiving a HO_REQ message from this node and successfully including it and its cluster into its own cluster. The message includes the length of the initial lease that was granted to the receiver by its new CH. The receiver first checks, if it is currently in the process of handing over its cluster. If so, it stops waiting for an answer to its HO_REQ message by canceling the *HandoverRequestTimeout* timer. After that it sets the message sender to be its new CH, becomes a CN and starts the lease mechanism. This is shown in Procedure 5.13.

Procedure 5.13 *OnHandoverAck*: Receiving a HO_ACK message

```
OnHandoverAck(SystemID $sender$, long $l$)
1: if $role$ = CH $\wedge$ $handover$ then
2:     stopTimer(HandoverRequestTimeout);
3:     $ch := sender$;
4:     SwitchRole(CN);
5:     startTimer(LeaseRenewalTimeout, $l$);
6: end if
```

HO_NACK Message

If a node receives a HO_NACK message, it first checks if it is currently in the process of handing over its own cluster to another node. If not, the message is omitted. Otherwise it stops the handover and resumes normal operation. In addition, it checks if its cluster is empty and if so, switches to the UN role. This is necessary, because a CH does not switch its role if its cluster is getting empty while trying to handover the cluster. While the handover is performed, the old CH has already handed the responsibility for its cluster to the new CH. Therefore, the removal of a node from its cluster is without consequence for it. If however, the handover fails, the responsibility falls back to the original CH.

Procedure 5.14 *OnHandoverNAck*: Receiving a HO_NACK message

```
OnHandoverNAck()
1: if $role$ = CH $\wedge$ $handover$ then
2:     $handover := false$;
3:     if $cluster = \emptyset$ then
4:         SwitchRole(UN);
5:     end if
6: end if
```

Handover Request Timeout

A *HandoverRequestTimeout* is similar to a *JoinRequestTimeout* besides that it is used for a handover. The timer is started when a CH sends a HO_REQ message. If no answer is received in time, a *HandoverRequestTimeout* occurs. The timeout is handled exactly like the reception of a HO_NACK (see Procedure 5.14).

5.7 Handling Communication Errors

For unicasts communication errors are corrected automatically, as we assumed reliable unicasts. For broadcasts this cannot be guaranteed. Therefore we have to cope with lost broadcast messages. Two types of messages are sent via broadcast, Announcement messages and LeaseRequest messages. In this section, we discuss how to handle the loss of these messages.

5.7.1 Lost Announcement Messages

If an Announcement message is lost, a node may not be detected by its neighbors or even worse may be removed from the neighborhood list because the cluster management falsely assumes that it has left communication range. Therefore it may not be joinable. To handle this, we increase the time that a node is included into the neighborhood due to an announcement message to a multiple of the announcement period. This makes removing a node that is no longer present slower and may lead to situations where a new node has never heard a left node while the others still have the left node in their neighborhood. Therefore the nodes may need longer to join. As an alternative, we could allow two nodes to join even if they are not in neighborhood all the time but it would be impossible to distinguish this from nodes moving in and out of their communication range continuously and was therefore discarded.

5.7.2 Lost LeaseRequest Messages

The loss of a LeaseRequest message can have two effects: first, the CH does not hear its CN, second, probing CNs do not discover the node. The first effect can be fixed by resending the lease request if the CH does not answer in time. It should be noted, however, that the CN has to try to renew the lease early enough to be able to retry sending the LeaseRequest message before the lease finally expires.

The second case is more complicated to handle. Note, that such losses cannot be tolerated by simply increasing the time that a node stays in the neighborhood list before it is removed. This would allow CHs and UNs to tolerate single message losses because only one of several LeaseRequest messages of a node must be received to keep the node in the neighborhood list.

For CNs in their probing phase, however, this does not solve the problem. A probing CN has only a limited time for detecting a node, namely the duration of its probing phase. The CN could very well miss its only chance to detect the node sending the LeaseRequest, without the possibility to receive the node's next LeaseRequest message because its probing could have already ended. The intuitive solution for this would be to prolong the probing phases of CNs such that it includes multiple lease requests. However, this would lead to longer activation times for all CNs and therefore lessen energy savings considerably.

Another possibility is to tolerate small differences in the neighborhood lists of a CH and its CN when comparing them at the end of the probing phase. Although this could lead to connectivity loss in certain cases, if the network is very lossy it could be beneficial to trade in a little connectivity for more robust clusters. Note, that this approach would also lessen the effect of lost Announcement messages.

The third possible approach is to retry a probing if the differences between the CH's and the CN's neighborhood lists are small. If the CH detects that its neighborhood differs only by a few nodes from that of its CN, it gives the CN another chance and lets it restart the probing instantly. Combined with a longer timeout before removing nodes from the neighborhood list, this allows the CN to detect nodes that it has missed the first time. In addition, the CN stays active only if needed, lessening the energy overhead compared to increasing the lengths of all probing phases.

5.8 Cluster Management in a Smart Environment

So far, we have seen how clusters are created and maintained in general. In the following, we briefly describe the behaviour of the cluster management protocol in Smart Peer Groups and Smart Environments. In a Smart Peer Group, all nodes in it are grouped into one cluster and the CH role is assigned to the node with the highest suitability. As nodes are battery-operated, the CH suitability will change over time and the CH role will be rotated resembling a round-robin pattern. If two Smart Peer Group come together, their CHs challenge each other, determine a winner and join their clusters. If some devices leave the Smart Peer Group, they detect the loss of their CH and restart clustering. Overall, the lifetime of all nodes is maximized. If a Smart Peer Group enters a Smart Environment, its CH challenges the Smart Environment's CH and in most cases looses, as the latter is most likely connected to the power grid. To reflect this, stationary nodes in the Smart Environment can be administrated to have a fixed and extremely high remaining lifetime, e.g., several years. By assigning specific remaining lifetimes, an administrator has complete control over the sequence in which infrastructure nodes are chosen as CH. Note, that no special precautions against a single point of failure are necessary. If the mediator fails, SANDMAN will detect this and elect another node, if possible, from the infrastructure. To allow members of the Smart Environment to detect services outside the environment, the mediator can use additional discovery protocols completely transparent to the nodes in the system. Note though, that the Smart Environment has to provide additional mechanisms to access such services.

5.9 Services Provided to Other Subsystems

The cluster management offers a number of services to the other SANDMAN subsystems. In this section, we describe these operations. We start by describing three operations that are offered two both subsystems. After that we present a number of operations that are provided specifically for the energy management subsystem.

GetCH

The GetCH operation accesses the local ch variable and returns its current value to the caller:

```
GetCH() returns SystemID
// Out SystemID : the identifier of the current CH
```

IsInCluster

The IsInCluster operation takes a single SystemID and checks whether the associated node is currently a member of the local nodes cluster:

```
IsInCluster(SystemID) returns Boolean
// In SystemID : the identifier of the node to check
// Out Boolean : whether the given node is in the cluster
```

If so, it returns true. Otherwise, it returns false. This functionality is used both by the Service-Manager (see Section 6.4.1) and the EnergyManager (see Section 7.3.4).

CheckCH

If a communication request was unsuccessful, other subsystems can use the CheckCH operation to instruct the cluster management to check whether the CH is still available.

```
CheckCH() returns Boolean
// Out Boolean : whether the CH of the requesting node
//               is still available
```

If CheckCH returns true, the CH is available and the requesting subsystem should retry its last operation. Otherwise, the Cluster Manager returns false and switches the node's role back to UN.

5.9.1 Services Provided to Energy Management

In addition to the services offerd both to the service and the energy management, the cluster management offers a number of operation specifically to the energy management. These operations are described in the following.

GetLeaseData

The GetLeaseData operation is used by the energy management to get all information necessary to piggy-back a lease renewal request in a SLEEP_ANC message (see Section 7.3.4).

```
GetLeaseData() returns data
// Out data : The data needed to for piggy-back a LEASE\_REQ message
```

SetLeaseRequest

When a CH receives a SLEEP_ANC message, the SetLeaseRequest operation is used by its energy management to send all information concerning a possibly embedded LEASE_REQ to the cluster management (see Section 7.3.4).

```
SetLeaseRequest(data)
// In data : The data received as a piggy-backed LEASE\_REQ message
```

The ClusterManager handles a call of this operation as if a LEASE_REQ message from the sender had arrived. The only difference to an ordinary LEASE_REQ message is, that the cluster management does not return a GRANT_LEASE message to the sender automatically. Instead, it grants the lease and returns a new data package to the energy management. This data package contains the data to send in the response message, send by the energy management.

GetLeaseLength

The GetLeaseLength operation is used to request the length of the lease of a CN at a CH's cluster management.

```
GetLeaseRequest(SystemID) returns length
// In SystemID : the identifier of the CN to check
// Out length : the remaining lease length of the requested CN
```

To do so, the operation takes one parameter, a SystemID denoting the CN whose lease length should be checked and returns the remaining lease length. If the requested node is not a member of the CH's cluster or the operation is called on a node currently not in CH role, the operation returns zero.

5.10 Related Work

In the following section we discuss existing work in the are of cluster management. Clustering is used for a multitude of different purposes, both in classical computer systems and MANETs, e.g., transmission management or hierarchical routing. A good general overview of different clustering algorithms is given, e.g., in [Ste01] and [YC05]. Since the main goals of our cluster management algorithm is to form stable clusters and to save energy by allowing nodes to be deactivated, we focus our discussion on clustering approaches which follow the same goals.

5.10.1 Stability-aware Approaches

Stability-aware clustering approaches (e.g., [CCCG97], [BKL01]) try to minimize the number of CH changes. To do so, they rely on event-driven cluster maintenance instead of periodic re-clustering. Our cluster management uses event-driven cluster maintenance, too.

Least Cluster Change

The Least Cluster Change (LLC) algorithm [CCCG97] relies on other algorithms, e.g., the lowest ID [EWB87] or the max-connectivity algorithm [GT95], to elect initial CHs. Once clusters are formed, LLC changes CHs only if absolutely necessary. If two CHs come in communication range with each other one of the CHs surrenders its role and the clusters are joined. In addition, if a non-CH has no CH in its communication range the node switches to CH role and tries to find members for its cluster. In LLC, a CH does not give up its role to a non-CH, even if this node would be better according to the initial clustering algorithm used.

MOBIC

MOBIC [BKL01] is a variant of LLC using the relative mobility of a nodes and its neighbors to choose CHs. To avoid nodes which pass by each other quickly to form a cluster, MOBIC introduces a so-called cluster contention interval. Only if two nodes are in contact after this interval they cluster themselves. The cluster contention interval resembles our cluster delay Δt_{join} closely. In comparison with LLC, MOBIC is able to reduce CH changes by approximately 33 percent [BKL01]. However, in contrast to our work, fairness is not an issue when selecting CHs. In addition, MOBIS tries to minimize CH changes. We aim at creating stable clusters, i.e., not only CHs changes but CNs staying in the same cluster for a long time. Lastly, MOBIC is not able to handle deactivated nodes, as neighbors are discovered by each node continuously. This is not possible for deactivated nodes as they are common for SANDMAN.

5.10.2 Energy-efficient Clustering

Energy-efficient clustering approaches (e.g., [XHE01], [CJBM02], [XBM$^+$03]) try to determine clusters such that non CHs can be deactivated to save energy. These approaches are typically used for routing and are thus trying to minimize the influence of deactivating nodes to the network's connectivity. Cluster stability is normally not an issue, however, leading to frequent cluster changes. Therefore, nodes can be deactivated only for up to a few seconds before their clusters must be redone. SANDMAN aims at deactivating nodes much longer, e.g., up to several minutes and thus needs much more stable cluster formations. In addition, existing energy-efficient clustering approaches do not support spreading wakeup times over time, like SANDMAN does, because they typically use a common wake time to find their neighborhood and form new clusters.

Geographical Adaptive Fidelity

Geographical Adaptive Fidelity (GAF) [XHE01] uses node locations to create a connected overlay network consisting of CHs. All other nodes, that are not a member of the overlay may deactivate themselves. To realize this, GAF divides the phyiscal environment into a virtual grid such that each node in one field of the grid can communicate with all nodes in all neighboring fields. Now, in each field, one node is chosen that joins the overlay, resulting in a connected overlay network. All other nodes can be deactivated temporarily. This approach has two main drawbacks: first, it requires location information about all nodes. SANDMAN is able to work without such information. Second, the resulting overlay can change quickly if nodes are very mobile. SANDMAN tries to cluster nodes according to their mobility pattern, not heir position, such that even fast moving nodes may be clustered, as long as they move together.

Cluster-based Energy Conservation

In contrast to GAF, Cluster-based Energy Conservation (CEC) [XBM+03] is independent of the availability of location information. Nodes are chosen as CHs, if their remaining lifetime is longer than that of their neighbors. SANDMAN uses a similar approach when comparing CH suitabilities. Clusters are formed such that the CH may communicate with all nodes in its cluster directly, but CNs may not be able to reach themselves directly. To communicate between clusters, special gateway nodes are used which connect multiple clusters. As CHs are not chosen according to their mobility, nodes must be reclustered regularly. Therefore, CEC allows only short deactivation durations. In addition, all nodes must be activated to form new clusters, therefore, no alternating deactivation periods as in SANDMAN are possible.

SPAN

In SPAN, each node individually observes its neighborhood to decide whether it should become a CH and join the overlay or deactivate itself and leave the overlay. This allows very flexible deactivation scheduling. However, to still be able to detect the current neighborhood of a node, SPAN relies on a lower layer to offer a continuous neighborhood detection. This is possible, because SPAN does not fully deactivate the nodes but only puts the the network cards into a power-save mode resembling that of IEEE 802.11. Therefore, the achievable energy savings are much lower than with SANDMAN who does not rely on any lower layer neighborhood detection.

5.10.3 Summary

There are a number of approaches that focus on either cluster stability or energy-efficient clustering. However, they do not support both at the same time. Approaches forming stable clusters rely on current neighborhood information and cannot handle temporarily deactivated nodes. Therefore, they cannot be used for SANDMAN. Still, we take some inspiration from these approaches, e.g., the cluster contention interval used by MOBIC. Existing energy-efficient clustering approaches dissolve their clusters regularly, leading to frequent cluster formations. This shortens the achievable deactivation times. In contrast to this, we try to detect stable system parts to allow longer deactivation times.

5.11 Conclusion

The responsibility of the cluster management is to create and maintain node clusters dynamically. To do so, CHs are elected using their remaining lifetimes as election criterium. Nodes are clustered if they have the same neighborhood for a certain time interval. This prohibits nodes passing by each other to from clusters for short times. Instead our cluster management aims at creating relatively stable clusters. This allows the energy management to deactivate nodes for long periods of time without considerable loss of discovery precision and recall. The cluster management forms the basis on top of which the service management and the energy management can operate. In the following sections we examine these subsystems in more detail.

6 Service Management

The service management subsystem is responsible for the management and distribution of service description data. It offers means to describe a service, to add it to the system, update, discover, and finally remove it. The service management is realized by the so-called *ServiceManager*, a BASE system service, as shown in Figure 4.4.

In this section, we describe the service management's structure and behavior. An overview of the issues that are discussed can be seen in Figure 6.1. The section is structured as follows: first, we give a short overview and discuss our service description language before the data structures used by the service management are presented. After that, we depict the service provider interface and the client interface. Finally, we describe how the data stored by the service management is maintained and how the service management interacts with the cluster management and the energy management.

6.1 Overview

Depending on the node's current role, i.e., if it is a CH, a CN, or a UN, the service management operates differently. If the node is a CH or UN, the service management is responsible for service discovery operations for itself and all nodes in its (possibly empty) cluster. It manages a local registry containing all services in the cluster and answers discovery requests from remote clients. In the CN mode, the service management has turned responsibility for all service discovery operations to its CH. Therefore it does not directly answer discovery requests and manages only its local services.

The overall organization of the service management and the interactions between the involved components can be seen in Figure 6.2. Whenever a service provider wants to offer a service,

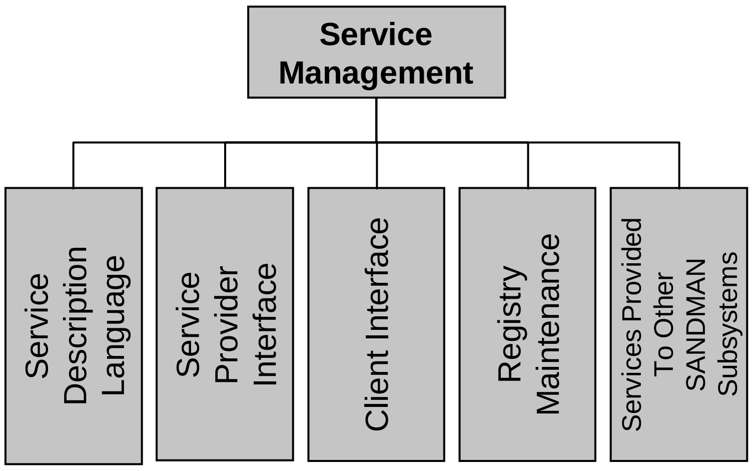

Service Management
Service Description Language
Service Provider Interface
Client Interface
Registry Maintenance
Services Provided To Other SANDMAN Subsystems

Figure 6.1: SANDMAN service management

it accesses the service provider interface (SPI) on its local ServiceManager (a). This interface offers means to register, update and remove a service. The ServiceManager stores this information in the *ServiceRegistry* (SR) (b). If a client wants to find a service, it accesses the client interface (CI) of its own local ServiceManager (c) and initiates a new discovery request. The ServiceManager handles this request and returns the results to the client.

Individual ServiceManagers do not operate in isolation. Instead, ServiceManagers interact with other ServiceManagers (d) to exchange information about offered and searched services with each other. As an example, the ServiceManagers of a CN and of its CH exchange all service information needed by the CH to act as a cluster-wide LUS. In addition, depending on the request, the CH's ServiceManager can handle a discovery request locally or may need to relay it to other ServiceManagers.

Finally, the service management interacts with the two other SANDMAN subsystems, the cluster management (e) as well as the energy management (f). As an example, the service management relies on the cluster management subsystem to set the correct node role. In addition, the service management offers the energy management the means to determine if any client has recently shown interest for the services of a given node. The energy management uses this information when determining whether a node should be deactivated.

6.2 Design Rationale

Before describing our service management in more detail, we give a short rationale of our main design decisions in the following section.

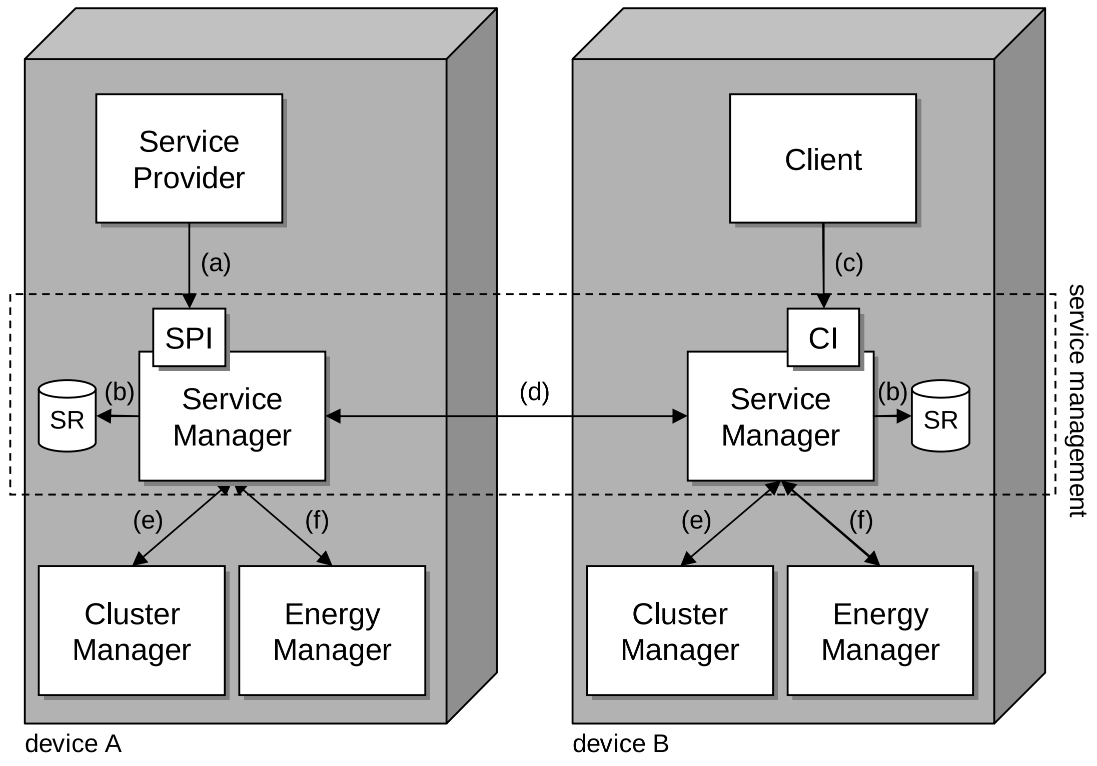


Figure 6.2: SANDMAN service management interactions

6.2.1 Service Availability

Most modern service discovery systems (e.g., [Sun01], [Mic00]) use a soft-state approach to guarantee that outdated service descriptors are removed from the system. To do so, they introduce so-called *leases*, that must be renewed regularly. If a lease expires, all service descriptors bound to this lease are removed. This allows to keep the system in a consistent state, even if service providers leave it silently. Clearly, since we can not assume that devices are always able to notify the rest of their SPG before leaving communication range or being powered down, SANDMAN must embody a similar mechanism to handle disappearing nodes.

A straight-forward approach to do so, would be include a lease mechanism for services into the SANDMAN service management. However, this would require service providers offering multiple services to extend multiple leases, leading to higher communication overhead. In addition, SANDMAN includes another subsystem, the cluster management, which already must handle disappearing nodes to react accordingly, e.g., recluster if a CH is lost. Therefore, we chose to separate the availability of services from that of nodes. Checking the availability of nodes is the responsibility of the cluster management, while the service management restricts itself to the handling of services. Both interact to make sure, that services offered by a leaving device are removed automatically. This allows to reuse the information gathered by the cluster management for the service management, thus saving additional overhead and energy. While a node is available, it will update the list of its services at its CH if needed, e.g., to remove a service descriptor. If the node becomes unavailable, the cluster management detects this and notifies the service management. The service management then can remove all services offered by this node from its registry. More information about the necessary cooperation between service and cluster management is given in Section 6.6.

6.2.2 Discovery Range

In normal discovery systems, a client has little or no control of how far its discovery request is propagated. As an example, in UPnP, each discovery request is multicast to all service providers in the network. In a wired network this might be not a real problem, because the routing infrastructure limits the multicast dissemination. In SPGs, a global search might lead to the whole network being flooded with discovery requests, consuming a considerable amount of bandwidth and energy.

To control the energy spend for a service discovery request, we introduce a new parameter, the so-called *discovery* or *lookup range*. The lookup range defines how far away the service discovery system should search for suitable services. Possible values are *local*, to search on the local device only, *cluster*, to search in the device's current cluster, and *full*, to specify that the discovery system should search the whole Smart Peer Group. An additional possibility would be to search a subset of the Smart Peer Group by specifying the number of hops that a lookup request should be forwarded. This would allow, e.g., to specify that the discovery system should search suitable services on all devices that are at most 3 hops away from the client device. However, from the systems's point of view this is mostly equivalent to searching the whole Smart Peer Group. The only difference is to use a hop-bounded multicast to disseminate the request instead of an unrestricted multicast. Therefore, this additional discovery range is omitted from our system so far. It can be easily added later, if needed.

Using the lookup range, a client can control the effort that the service discovery system puts into finding a suitable service. A local search can be done without remote interaction. A cluster-wide search is realized by contacting the client's CH (possibly itself). Only in case of a full search the request needs to be distributed to the whole Smart Peer Group. An additional effect of the lookup range is that the client indirectly specifies how stable its connection to the service should be. If a service is located in the same cluster, the client's and the service's nodes most probably have similar mobility patterns. Therefore, a client may prefer a service in the same cluster over one in another cluster. By using a series of lookup requests with expanding lookup ranges a client can search for suitable services with increasingly higher effort and decreasingly stable connections between client and service.

6.2.3 Service Description Language

A service description language (SDL) is used to define an offered or needed service. Although different languages for offered and needed services are possible, using the same language simplifies both the system usage and the discovery process. In the past, different service description languages have been defined, e.g., the Web Service Description Language (WSDL) [CMRW06], DAML+OIL [CvHaDLM$^+$01] or the CORBA Interface Description Language [Obj04]. SANDMAN can be combined with all these languages. In our realization, SANDMAN is integrated with BASE, written in Java. Therefore, we used Java as the basis for our service descriptions for simplicity.

We model a service as a 4-tupel, consisting of a globally unique service identifier (*id*), a provider selected name (*name*), a number of service types (*type[]*) and a number of non-functional attributes (*attr[]*) in the form of name-value pairs. This conforms to the way most current service discovery systems model their services (e.g., [Sun01], [Mic00]). Therefore, it's easy to use SANDMAN with one of these systems. More complex service models, e.g., models using a semantical service description like DAML+OIL, can be integrated in the future but are not examined in this work.

The globally unique identifier is used to identify a specific service. The globally unique service name is specified by the service provider or rather the service developer. It can be used to select a specific service by specifying its name. Note, that it is the service developers responsibility to create a globally unique name. SANDMAN does not provide any mechanism to ensure that a name is unique in the system. At system runtime, in addition to the name the service identifier could be used to select a specific service. However, SANDMAN does not guarantee that a service is bound to the same identifier at any point in time. If the service is removed from the system and offered again later, its identifier may change. Its name, however, stays the same. Therefore, the identifier is used by the system, e.g., the BASE micro-broker, to identify a given service at runtime, while the name should be used to identify a specific service by developers. The service types specify the functional interface of the service, i.e., the names and parameters of the service's operations. We use the Java type system to model such service types. The service attributes can be used to specify additional non-functional properties of a service.

Note, that although we offer means to change service descriptions over time, we do not support dynamic attributes, like e.g., offered by CORBA. Such attributes are evaluated at discovery time by the service provider. In SANDMAN, the service provider may not be active at the time of a discovery. Therefore the discovery would have to be delayed to enable the provider evaluating dynamic attributes. As an alternative, some dynamic attributes could be evaluated by the CH following instructions from the service providers. However, due to the CH not knowing the providers current state, this is not possible in general. We therefore restrict our system to static attributes that can be updated by the service provider at any time, mimicking dynamic attributes to a certain extend.

6.3 Data Structures

The ServiceManagement introduces four data structures: the *ServiceID* is a globally unique service identifier. The *ServiceDescriptor* specifies the properties of a service. The *ServiceRegistry* is the central data store for all data concerning local and remote services. Finally, the ResultSet is used to exchange the result of a discovery request between system entities. We present these data structures briefly in the following.

6.3.1 ServiceID

To identify a specific service in the system, a globally unique service identifier is needed. To create such identifiers, SANDMAN relies on BASE. Using the node's unique identifier and a local counter, the BASE micro-broker can be queried for new unique identifiers. SANDMAN takes these identifiers and creates *ServiceIDs* from them. To model this behavior, we introduce the operation `createServiceID` which returns a new unique ServiceID.

6.3.2 ServiceDescriptor

To realize the service model presented in Section 6.2.3, SANDMAN uses so-called *ServiceDescriptor* records. A ServiceDescriptor specifies the properties of a service and contains the name, types and attributes of a service:

```
ServiceDescriptor = ( name, type[], attr[] )
// name   : name of the service
// type[] : types of the service
// attr[] : attributes of the service as name, value pairs
```

Each element of a ServiceDescriptor may be null or empty. This is used to specify the properties of a searched service. As an example, a client may specify only the name of a service that it searches or it may specify its types and some additional attributes.

6.3.3 ServiceEntry

While a ServiceDescription contains all information needed to offer and to search a service, more knowledge is needed to access and use a service. The service management stores this information in ServiceEntrys. Each ServiceEntry contains all information for one service. More specifically, it contains the SystemID of the node offering the service, a list of all plugins that can be used to access this node, information about the node's deactivation schedule, e.g., if the node is currently deactivated, the ServiceID of the service and the service's ServiceDescriptor:

```
ServiceEntry = ( provider, plugins[], schedule, service, descriptor )
// provider   : SystemID of the node offering the service
// plugins[]  : plugins that can be used to access this node
// schedule   : the node's deactivation schedule
// service    : ServiceID of the service
// descriptor : ServiceDescriptor of the service
```

This is the complete set of information available for each service in SANDMAN. Note, that in our implementation, the ServiceEntry contains only references to the node centric data as this data is the same for all services offered by one node. This results in much smaller messages when transmitting a lot of ServiceEntrys, e.g., when a node enters a new cluster as shown later in this chapter.

6.3.4 ServiceRegistry

The *ServiceRegistry* is the central data structure of the ServiceManagement. It contains all data needed to discover and access services and consists of a set of ServiceEntries. Regardless of a node's current role, its ServiceRegistry contains all local services, i.e., all services that are offered by the device itself. In addition, if the device on which the registry is placed is currently a CH, it also contains all remote services that are offered by other devices in its cluster. To use the ServiceRegistry, it offers five procedures: Contains, Add, Get, Remove, and GetAccessTime, which we describe in the following.

Contains(ServiceID) returns Boolean The *Contains* operation checks if a service with a given service id is already contained in the ServiceRegistry. If so, it returns `true`, otherwise `false`.

Add(ServiceEntry) Using the *Add* operation a new service is added into the ServiceRegistry. Add takes as its single parameter the ServiceEntry containing all information about the service.

Get(SystemID, ServiceDescriptor) returns ServiceEntry[] The *Get* operation searches the ServiceRegistry and returns an array containing all ServiceEntrys for services that are offered by the node with the specified SystemID and fulfill the given ServiceDescriptor. Note, that the first parameter may be omitted by specifying *NULL* as the SystemID. This returns all matching services in the ServiceRegistry, regardless of the node offering them. When including a ServiceEntry into its return parameter, *Get* stores a current time stamp and links it internally to the ServiceEntry. This information is used by the GetAccessTime operation.

Remove(ServiceID) To end offering a service and remove its ServiceEntry from the ServiceRegistry, the *Remove* operation can be called. The service to be removed is specified by its ServiceID, which is passed as the operation's single parameter.

GetAccessTime(ServiceID) The GetAccessTime operation searches the ServiceRegistry for all services offered by the node with the specified identifier. After that, it accesses the time stamps associated with their ServiceEntrys and returns the most current time stamp found. This way, the operation returns the last time that a service offered by the specified node was searched in the ServiceRegistry. This functionality is used by the `LastDiscovery` operation discussed in Section 6.7.2 in order to determine if a given CN can be savely deactivated.

6.3.5 ResultSet

A ResultSet is used to transmit the results of a lookup request to other nodes and the client. It contains data about all services fulfilling the request, the nodes offering them, and how to access the services. A result set consists of ServiceEntrys, each containing all information for one service. ResultSet offers two operations to add ServiceEntrys to it: *Add* and *Get*, which we describe in the following.

Add(ServiceEntry[]) To add ServiceEntries to it, the ResultSet offers the *Add* operation. *Add* takes an array of ServiceEntrys as returned by the *Get* operation of the ServiceRegistry and adds them to the ResultSet.

Get() returns ServiceEntry[] To access the ServiceEntrys included in a ResultSet, the *Get* operation can be used. Get returns an array containing all ServiceEntrys in the ResultSet.

6.4 Service Provider Interface

SANDMAN offers a local service provider interface. Using this interface, a service developer can register a new service, update a service registration with new information and remove an existing service.

6.4.1 Registration

Registration of services is done by calling the `Register` operation (shown in Procedure 6.1) at the local ServiceManager with a ServiceDescriptor record and a local reference to the service.

Procedure 6.1 *Register*: Register a service

```
Register( ServiceDescriptor d, Ref s) returns ServiceEntry
id := createServiceID();
plugins[] := BASE.PluginManager.GetAll();
schedule := EnergyManager.GetSchedule();
entry := new ServiceEntry(this, plugins, schedule, id, d);
ServiceRegistry.Add(entry);
BASE.ObjectRegistry.Add(id, s);
if role = CN then
  usend({ch,ServiceManager}, REGISTER, {entry});
  if not successful then
    if ClusterManager.CheckCH() = true then
      retry remote registration;
    else
      abort;
    end if
  end if
end if
return entry;
```

`Register` assigns a new ServiceID to the service and adds a new ServiceEntry entry into the ServiceRegistry. To obtain the necessary information for the ServiceSet, the ServiceManager queries the BASE plugin manager to get the plugin information and the energy management to get the node's deactivation schedule.

After that, it accesses the BASE ObjectRegistry and binds the new ServiceID to the service implementation. This allows BASE to forward incoming messages to the service. Service providers could access the ObjectRegistry directly to include this binding in BASE. However, letting the ServiceManager perform the binding creates a leaner, more convenient registration interface. After this, the procedure checks if the node is currently operating as a CN. If so, it relays the service registration to the current CH to update the cluster-wide ServiceRegistry maintained there.

Normally, this ends the Register operation, leaving an open question. What should happen, if the remote registration at the CH fails? The ServiceManager can retry registration. However, as we assumed reliable unicast communication, the only reason that the registration could have failed is that the clustering has changed, e.g., because the CH is out of communication range or because it is no longer the sender's CH. As said before, in SANDMAN it is the cluster management's responsibility to detect such situations and change the node role accordingly. Therefore, if the registration fails, the service management notifies the cluster management which then checks if the CH is still available. If so, it notifies the service management, which retries the registration. If not, the cluster management leaves the cluster and notifies the rest of the system of this fact.

When receiving a REGISTER message (see Procedure 6.2), the CH checks if it is currently a CH and if the sender of the registration message is a CN in its cluster. To do so, it contacts the ClusterManager and calls its *IsInCluster* operation. This operation takes the identifier of the node in question and returns true if the node is in this CHs cluster, false otherwise. If both conditions are fulfilled, the CH includes the new service into its local ServiceRegistry and sends a REGISTER_ACK message to its CN. Otherwise, it sends a REGISTER_NACK message, notifying the original sender of the fact, that the registration was not successful. This triggers the `not successful` condition in Procedure 6.1 on line 9.

Procedure 6.2 *OnRegister*: Receiving a REGISTER message

OnRegister(SystemID sender, ServiceEntry e)

```
1: if $role$ = CH $\wedge$ ClusterManager.IsInCluster(sender) then
2:     ServiceRegistry.Add(e);
3:     usend({sender,ServiceManager}, REGISTER_ACK);
4: else
5:     usend({sender,ServiceManager}, REGISTER_NACK);
6: end if
```

6.4.2 Update

If the state of a service changes, e.g., in case a GPS receiver becomes available because the user moves out of a building, the service provider may choose to update the service description. To do so, it calls the `Update` operation shown in Procedure 6.3 and passes a ServiceID identifying the service to change and a ServiceEntry containing the new data about the service. `Update` checks if the given service identifier is valid and updates the corresponding ServiceEntry in the local ServiceRegistry. If the node is a CN, the procedure then sends an UPDATE message to the CH specifying the new ServiceEntry. If this update fails, the system behaves as in the registration case. If no ServiceEntry for the given ID is found, the operation does not change the state of the ServiceRegistry.

When receiving an UPDATE message (see Procedure 6.4), the receiver first checks if it is the sender's CH and if its ServiceRegistry contains an entry for this service. If not, it returns an UP-

Procedure 6.3 *Update*: Update a service registration

```
Update( ServiceID id, ServiceEntry entry )
if ServiceRegistry.Contains(id) then
  ServiceRegistry.Remove(id);
  ServiceRegistry.Add(entry);
  if role = CN then
    usend(ClusterManager.GetCH(), ServiceManager, UPDATE, {id, entry});
    if not successful then
      if ClusterManager.CheckCH() = true then
        retry remote update;
      else
        abort;
      end if
    end if
  end if
end if
```

DATE_NACK message to the sender, triggering the not successful condition in Procedure 6.3, line 6.3. Otherwise it updates the information in its registry and sends an UPDATE_ACK message to the sender.

Procedure 6.4 *OnUpdate*: Receiving an UPDATE message

```
OnUpdate(SystemID sender, ServiceID id, ServiceEntry e)
if role = CH $\wedge$ ClusterManager.IsInCluster(sender) $\wedge$ ServiceRegistry.Contains(id) then
  ServiceRegistry.Remove(id);
  ServiceRegistry.Add(e);
  usend({sId,ServiceManager}, UPDATE_ACK);
else
  usend({sId,ServiceManager}, UPDATE_NACK);
end if
```

6.4.3 Removal

To remove an offered service from the system, a service provider calls the operationRemove operation (see Procedure 6.5) passing the unique service identifier that it has obtained from the original service registration. The service management checks if the ServiceID is valid and removes the corresponding service from its local ServiceRegistry. In addition, a CN sends a REMOVE message to its CH to update its registry, too.

Procedure 6.5 *Remove*: Remove a service registration

```
Remove( ServiceID $id$ )
1: if ServiceRegistry.Contains(id) then
2:     ServiceRegistry.Remove(id);
3:     if $role = $ CN then
4:         usend($ch$, ServiceManager, REMOVE, {this, id});
5:     end if
6: end if
```

Note, that if the CH is not accessible at removal time, i.e., the sending of the REMOVAL message fails, the service management can savely ignore this fact. In contrast to other systems (e.g., Jini) the service management subsystem offers no soft-state approach for lease-based service removal. Instead, the detection of a missing CH is the responsibility of the cluster management sub system and not of the service management. When the cluster management detects that the CH has become unavailable, it notifies the service management of this fact, enabling it to take further actions. See Section 6.6 for more information on this.

6.5 Client Interface

In addition to the service provider interface, SANDMAN offers a client interface to search for specific services. SANDMAN offers a rather lean client interface. This is due to two facts: first, in order to enable portable client code, the interface should be identical on all kinds of devices. Therefore, it must be applicable for small resource poor devices. Second, the design of a sophisticated client interface is beyond the focus of this dissertation. Other interfaces with additional features, e.g., to sort results or restrict the number of returned results, can be realized on top of SANDMAN.

6.5.1 Initiating a Lookup Request

To search a service, a client calls the `Lookup` procedure shown in Procedure 6.6.

`Lookup` takes two parameters, a ServiceDescriptor (d) and a lookup range (r), as discussed in Section 6.2.2. The ServiceDescriptor specifies the properties of the searched service(s). By including entries for some of its parts and leaving other parts empty, the ServiceDescriptor can be used to specify different search types. As an example, it may specify that the searched service should be of a specific type to perform a yellow page search or it may specify the service's name to initiate a white page search.

Procedure 6.6 *Lookup*: Search for services

```
Lookup( ServiceDescriptor $d$, Range $r$ )
 1: if $r$ = local then
 2:   ResultSet.Add(ServiceRegistry.Get($this$, $d$));
 3: else if $r$ = cluster then
 4:   if $role = \mathbf{CH} \vee role = \mathbf{UN}$ then
 5:     ResultSet.Add(ServiceRegistry.Get(NULL, $d$));
 6:   else
 7:     ResultSet.Add(ServiceRegistry.Get($this$, $d$));
 8:     $id$ := GetNewRequestID();
 9:     usend($ch$, ServiceManager, LOOKUP_REQ, $\{id, d\}$);
10:     wait for LOOKUP_RESP from $ch$;
11:     ResultSet.Merge(ResultSet contained in LOOKUP_RESP);
12:   end if
13: else if $r$ = full then
14:   ResultSet.Add(ServiceRegistry.Get(NULL, $d$);
15:   $id$ := GetNewRequestID();
16:   msend(DISC_MC_GROUP, LOOKUP_REQ, $\{id, d\}$);
17:   start timer with length $T_{disc}$;
18:   while not timeout do
19:     wait for LOOKUP_RESP from * until timeout;
20:     ResultSet.Merge(ResultSet contained in LOOKUP_RESP);
21:   end while
22: end if
23: return ResultSet;
```

Note that a client cannot search for a service by its ServiceID. This is the case because the ServiceID can be used directly to access a service without the need to search for it. To do so, the client creates an Invocation, addresses it to the ServiceID and hands it to the BASE micro-broker. The micro-broker then uses its local plugins to forward the Invocation to the service. However, as already stated, the identifier of a service may change during runtime of the system. Therefore, developers normally should not use the ServiceID to directly specify the service they want to access. Instead, they should use the service name to do so.

In case of a full lookup, all CHs and UNs in the Smart Peer Group must be queried. To do so, the service management multicasts the lookup as a LOOKUP_REQ message to a special discovery multicast group DISC_MC_GROUP. Clearly, this multicast could be replaced with a series of unicasts to all available LUSes, i.e., all CHs and UNs. However, this would introduce the need for a special discovery protocol for such nodes. This approach is for instance taken in

Jini. However, Jini assumes a rather static system in which LUSes change rarely. In our system model, existing LUSes may enter the system at any time and new LUSes are elected continuously. Therefore we would have to discover LUSes continuously, which would introduce a considerable overhead and could cause new sources of inaccuracy, e.g., if a new LUS has not yet been discovered. Therefore, we chose the more robust approach of distributing the lookup request to all LUSes in the Smart Peer Group using multicast.

After sending the request message, the client has to wait for suitable answers. In case of a cluster-wide request of a CN, the client will receive a single answer from its CH. If communication with the CH fails, `Lookup` signals the cluster management and waits for it to decide whether the CH is still available, analogous to the operations in the service provider interface. This mechanism is omitted in Procedure 6.6 to make the operation easier to understand.

If a full lookup is performed, the client may receive answers from a number of CHs and UNs in the Smart Peer Group. It cannot decide when all answers have been received. Therefore, it uses a timeout mechanism to wait for responses for a given time interval T_{disc}. After this time, the ServiceManager merges all results and returns them to the client. The value of T_{disc} could be given by the client. A high T_{disc} might result in a higher discovery recall, a low T_{disc} results in a lower discovery latency. Therefore, by setting T_{disc} the client can provide its own trade off between these values. However, it is difficult for the client to estimate the concrete impact of a specific T_{disc} on its discovery result. Therefore, SANDMAN uses a fixed value for T_{disc} to provide a leaner client interface.

Now, the client can select the service suited best for its goals, e.g., the service that awakes first. If the information contained in the ResultSet indicates that a given service is offered by a currently deactivated node, the client waits for the service's node to be activated again before contacting it. Note, that during this time, a client can decide to deactivate itself to save additional energy. In addition, clients can use a randomized backoff delay before contacting the service to avoid collisions with other clients.

6.5.2 Answering a Lookup Request

Upon reception of a LOOKUP_REQ message, the `OnLookupReq` operation shown in Procedure 6.7 is executed.

The receiver first checks if it is currently responsible for answering lookup requests. This may be the case, if the receiver has recently changed its role to CN but has not yet left the multicast group. If the node is a CN, it abort the procedure. An UN creates a new ResultSet containing all its local services matching d. This ResultSet is sent back to the requesting ServiceManager located on the client. A CH additionally checks if the requesting ServiceManager is located on a node in its cluster. If so, it removes all services locally offered by this node from the ResultSet

Procedure 6.7 *OnLookupReq*: Receiving a LOOKUP_REQ message

```
OnLookupReq( SystemID $sId$, RequestID req, ServiceDescriptor $d$ )
 1: if $role = $ CN then
 2:     abort;
 3: else if $role = $ UN then
 4:     ResultSet.Add(ServiceRegistry.Get($this$, $d$));
 5:     usend($sId$, ServiceManager, LOOKUP_RESP, {req, ResultSet});
 6: else
 7:     ResultSet.Add(ServiceRegistry.Get(NULL, $d$));
 8:     if ClusterManager.IsInCluster($sId$) then
 9:         ResultSet.Remove(ServiceRegistry.Get($sId$, $d$));
10:     end if
11:     usend($sId$, ServiceManager, LOOKUP_RESP, {req, ResultSet});
12: end if
```

before sending it. This reduces the communication amount, as these services can be queried at the requesting node's local ServiceRegistry.

6.6 ServiceRegistry Maintenance

Besides service providers adding, updating and removing services by calling the according operations in the service provider interface, the state of the ServiceRegistry is highly dependent on the clustering state of the node. As an example, if a CH looses one of its CNs, all services offered by this node must be removed from the ServiceRegistry. To maintain the integrity of the ServiceRegistry, the service management has to take steps in the following clustering-related events:

- the node is a UN or a CN and joins a new cluster or changes to a new CH,
- the node becomes a CH itself,
- the node is a CH and a new node joins its cluster,
- the node is a CH and a node leaves its cluster,
- the node is a CH and becomes unclustered, or
- the node is a CH and joins another cluster, i.e., it switches to role CN.

The ServiceManager relies on the cluster management to notify it in case one of these events occur. To be notified, the ServiceManager subscribes to a number of events fired by the cluster management. We describe these events, when they occur and the actions necessary when they do in the following section.

6.6.1 Joining a Cluster

If a node joins a cluster, the new CH has to be notified about all previously registered services. Otherwise, the CH's ServiceRegistry misses services and the CH can not perform its LUS function effectively. The intuitive approach to do so would be to subscribe for a ClusteredEvent at the CM. This event is created whenever a node enters a new cluster, more specifically, whenever it changes its role from UN to CN. On receiving a ClusteredEvent, the ServiceManager registers all its services at the new CH by sending REGISTER messages for all its services. However, this leads to a rather complex system, because nodes do no longer change their roles atomically. Instead, the ServiceManager has to distinguish between a CN that has already registered all its services and can therefore stop answering lookup requests and a CN that has not yet done so and must still answer lookup requests. In addition, the ServiceManager must handle cases in which a cluster was successfully joined but service registration failed. This can happen, if the newly joined CH has decided to give up its cluster. The ServiceManager must specify what should happen in such cases. One possibility would be to handle such failures analogous to failed single registrations as discussed before.

A much more simple approach is to unite both cluster join and services registration into a single step. This allows a clean switch between node roles and saves resources because only one message must be send. To do so, the ServiceManager offers two new operations `GetServices` and `AddServices` to the cluster management. Before joining a cluster, the cluster management calls the `GetServices` operation and gets all data about services offered by this node. It includes this data into the join message, effectively piggy-backing the service registration. When receiving such a join message, the cluster management separates all service information and forwards it to its local ServiceManager by calling the operation `AddServices`.

6.6.2 Cluster Handover

If a CH decides to hand its cluster over to another node, either another CH or one of its CNs, SANDMAN must ensure, that the information contained in the old CH's ServiceRegistry is transferred to the new CH, according to the handover mechanism presented in Section 5.2.8. The situation resembles that of a node joining a cluster, as discussed in the previous section, closely. Again, one possibility would be for the ServiceManager to register at the ClusterManager for a event notifying it of the handover. However, this would again lead to a complicated system behavior, as the old CH must continue its work as the cluster's LUS until the contents of its ServiceRegistry is completely transferred. In addition, a large number of REGISTER messages would have to be send, as a CH will likely manage a large number of services. Therefore we again decide to combine both the cluster handover and the LUS handover in a single message. The ServiceRegistry content is piggy backed on the HO_REQ message. The ClusterManager uses the `GetServices` and `SetServices` operations as described in Section 6.7.1 and Section 6.7.1 to get and set the necessary service information.

Note, that handing over its cluster implies loosing its CH role and must be handled as such by the initiating CH as described in Section 6.6.4.

6.6.3 Loosing a Cluster Member

If a cluster member is lost, a *ClusteredNodeLost* event is created on the CH by the cluster management. On receiving such an event, the service management removes all data concerning services offered by the lost node from its service registry.

6.6.4 Loosing the Cluster Head Role

If a CH becomes unclustered or joins another cluster, it has to remove all services in its ServiceRegistry that are not offered by itself. To do so, the ServiceManager registers itself for a ClusterHeadRoleLost event at the ClusterManager. When the node changes its role from CH to UN or CN, the ClusterHeadRoleLost event is fired and the ServiceManager removes all entries from its ServiceRegistry that are offered by other nodes.

6.6.5 Changing Plugin Availability

Despite the necessary actions when changes in the clustering status of a node occur, the ServiceManager has to adapt the ServiceRegistry to another set of events, namely changes in the availability of a node's plugins. Whenever a new plugin becomes available on a CN or an existing one is removed, the CN's entry in the CH's ServiceRegistry must be updated accordingly. To do so, the ServiceManager registers itself for two additional events at the BASE PluginManager: *PluginAdded* and *PluginRemoved*. If one of these events occurs, the set of plugins offered by the CN has changed. Therefore, the CN's ServiceManager retrieves the new set of plugins and updates the CN's entries in its CH's ServiceRegistry. To do so, the CN's ServiceManager calls the Update operation as defined in the ServiceProvider interface repeatedly. Note, that a more efficient possibility would be to introduce a special Update message to the CH, updating all services offered by a certain CN at once. However, plugins change rarely. Therefore, we omitted introducing a special message for this case to keep the system smaller and less complex.

6.7 Services Provided to Other Subsystems

As already discussed, the service management not only relies on other SANDMAN subsystems for its proper operation but offers a number of operations to both the cluster management and the energy management, too. These operations are described in the following section.

6.7.1 Services Provided to Cluster Management

The ServiceManager offers the cluster management two services, modeled as the two operations `GetServiceData` and `SetServiceData`. These operations are described in the following.

GetServiceData

As described before, when a node joins a cluster or hands its own cluster over to another node, the information contained in the ServiceRegistry must be transferred to the new CH. To do so, this information is piggy-backed on a JOIN_REQ or HO_REQ message send by the ClusterManager. The GetServiceData operation is used by the ClusterManager to get all information needed to do so from the ServiceManager.

SetServiceData

The SetServiceData operation is the pendant of the GetServiceData operation discussed before. SetServiceData is called by the ClusterManager to push all service related information that it received in a JOIN or HANDOVER message into the ServiceManagement. The ServiceManagement accesses this information and includes it in its ServiceRegistry.

6.7.2 Services Provided to Energy Management

In addition to the services provided to the cluster management, the service management also offers a number of services to the energy management. These are modelled as three operations, `LastDiscovery`, `UpdateSchedule`, and `GetServices`, which are discussed in the next sections.

LastDiscovery

The LastDiscovery operation (see Procedure 6.8) is used by the energy management to determine whether a node should be deactivated or not. See Section 7.3.1 for a discussion of this.

The operation takes a single parameter, a SystemID, and returns a time interval value. This interval denotes the period of time since the last time that a service offered by the node with the given SystemID was returned in a lookup reply by the local ServiceManager.

Procedure 6.8 *LastDiscovery*: Compute last time a node's services were discovered

```
LastDiscovery( SystemID $i$)
1: $t$ := SystemRegistry.GetAccessTime(i);
2: $\Delta t := t_{now} - t$;
3: return $\Delta t$;
```

UpdateSchedule

When a CN is deactivated, its ServiceEntry must be updated with a new deactivation schedule. To do so, the UpdateSchedule operation can be used. UpdateSchedule takes the SystemID *i* of the node which services should be updated and the new schedule *schedule* to set.

Procedure 6.9 *UpdateSchedule*: Update a node's deactivation schedule

```
UpdateSchedule(SystemID $i$, $schedule$)
1: entries[] := ServiceRegistry.Get(id, new ServiceDescription(*));
2: for all e $\in$ entries do
3:     e.schedule := schedule;
4:     ServiceRegistry.Remove(e.service);
5:     ServiceRegistry.Add(e);
6: end for
```

GetServices

The EnergyManager uses the services offered by a node to calculate this node's deactivation period. To access the information what services are offered by a node, the EnergyManager uses the ServiceManager's GetServices operation. This operation takes a SystemID, denoting the node offering the services and returns an array of ServiceEntries containing all services offered by this node. To implement the GetServices operation, the ServiceManager accesses its ServiceRegistry and calls its Get operation with the given SystemID as the first parameter and an empty ServiceDescriptor as the second parameter.

6.8 Conclusion

The service management subsystem allows to publish service offers and specify client needs in order to mediate between services and clients. To do so, it stores service related information in the ServiceRegistry and offers different interfaces to interact with other entities in the system. To allow service providers to add, update and remove services, it offers a service provider interface. To let clients search for services, a client interface is provided. In addition, it interacts with remote ServiceManagers to exchange information about services and clients. Lastly, the service management offers an interface to the other two SANDMAN subsystems.

7

Energy Management

The energy management is responsible for deciding when to deactivate and when to reactivate a node in order to save energy. Our goal is to deactivate nodes while they are not needed. Therefore, we have to provide means to identify idle nodes. When a node is needed, it should be activated as soon as possible to minimize delays for clients. Otherwise, the system performance could be degenerated. In addition, clients consume energy while waiting, possibly negating the achieved energy savings. Finally, deactivating and activating nodes consumes energy itself. Therefore, it may be beneficial to stay active in some situations, if the deactivation duration is too small.

In the following, we describe how the SANDMAN energy management decides when to deactivate a device and when to reactivate it. We start with a short overview and a design rationale, discussing the basic design decisions taken. After that, we describe two approaches to schedule deactivation periods. The first basic approach uses only node specific information to determine a single CN's transitions to and from the low-power mode. The second extended approach uses information about the whole cluster to schedule the transitions of all cluster members. We close the chapter with a short conclusion.

7.1 Overview

The energy management is central to the saving of energy in SANDMAN. The cluster management dynamically forms and maintains node clusters. The service management is able to publish and discover services. However, these components do not save any energy by themselves. Instead, it is the energy managements responsibility to determine when a node should be deactivated in order to save energy and to balance the achievable energy savings with the resulting delays. To do so, it interacts with all other components to get information about the current state of the node and the cluster. The overall organization of the energy management and its interactions with the other system components are shown in Figure 7.1. The local EnergyManager continuously monitors the state of its device, using information provided by the BASE Micro-broker (a), the ClusterManager (b), the ServiceManager (c), and a local idle timer (d), which is discussed later in more detail. When it finds that the node can be deactivated savely (see Section 7.2.1), it contacts the remote EnergyManager of its current CH and negotiates the deactivation with it (e). After this negotiation is finished successfully, the EnergyManager deactivates the local device temporarily.

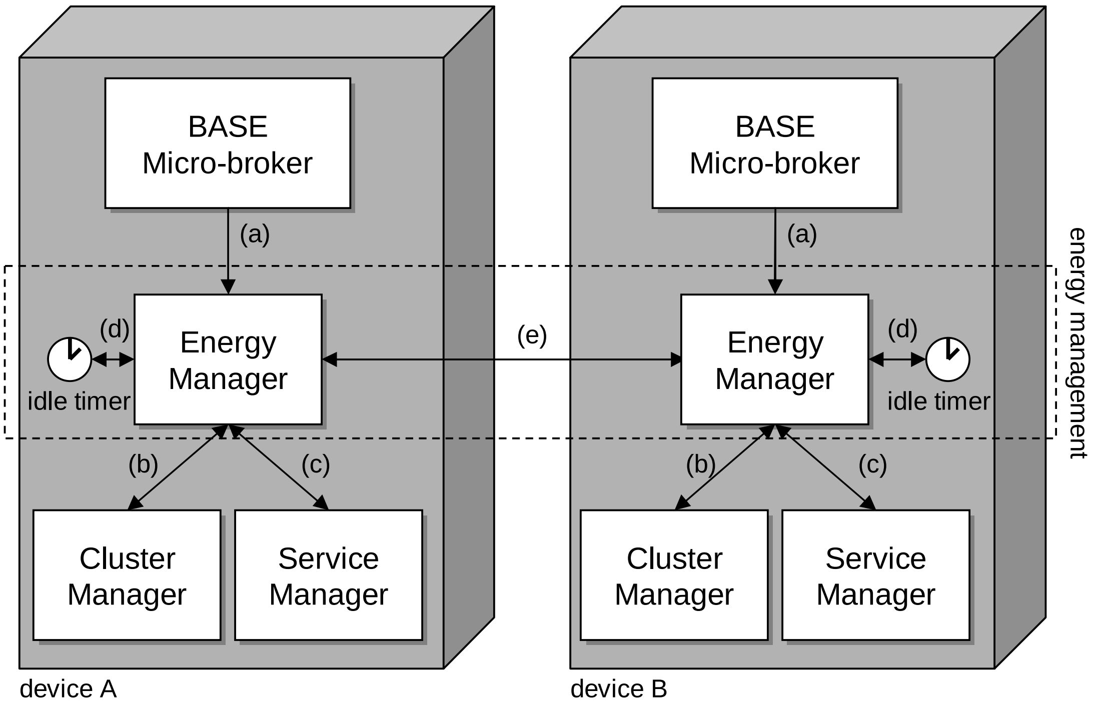


Figure 7.1: SANDMAN energy management interactions

7.2 Design Rationale

When designing an energy management for SANDMAN, two main questions arise. First, when should we deactivate a device, i.e., switch it into its low-power mode. Second, when should we reactivate the device, i.e., switch it back to its fully operational normal mode. These two questions are discussed in more detail in the following before we present our two approaches for energy management in SANDMAN.

7.2.1 Node Deactivation

The question when to switch a node into a mode with low-power consumption and reduced functionality is known as the transition problem. An algorithm to solve the transition problem is called a transition strategy [LS98] or power management policy [SBM99]. Different transition strategies have been developed in the research area of Dynamic Power Management (DPM). In DPM, the operating system executes a transition strategy to identify and power down unneeded device components, e.g., a hard disk or a display, until they are needed again. To do so, different approaches exist. Following [LSM99] we distinguish between heuristic approaches, like, e.g. strategies using inactivity thresholds [GBS+95], and predictive approaches using, e.g., Continuous-Time Markov Decision Processes [QP99]. A comprehensive comparison of transition strategies can be found in [LCS+00] and [BdM00].

SANDMAN can make use of any such transition strategy. It extends the chosen strategy by adding application-level information in the decision process. In this work, we use a fixed inactivity threshold strategy, because such an approach can be realized very efficiently with scarce device resources. More complex approaches could be integrated in the future.

7.2.2 Node Reactivation

Once a node has transitioned to its low power mode, it stays in this mode until it is reactivated. In typical DPM approaches, this reactivation is triggered by accessing the node, i.e. reactively when the node is needed. In the rare case, that an automatic activation is not possible, e.g., because a device is deactivated fully, the user can be expected to manually fix this by pressing the power button and manually switching the device back on. This behavior is not acceptable in pervasive computing. While a user could be required to manually activate some prominent devices like a PDA or laptop to allow its usage, it is not feasible to assume this for all devices, like small sensor nodes, etc. Therefore, we have to automatically activate needed devices. Shih et al. propose an out-of-band signaling to reactivate remote nodes [SBS02]. However, this requires to add special communication hardware to all nodes that must be active at all times. We do not assume the presence of such a hardware to stay independent of the actual communication interface used. Therefore, a node is reactivated by an internal watchdog timer after a given time. Such timers exist in most modern processing units and can be operated at very little cost (see e.g., [Atm06]). This leads to a proactive reactivation strategy, as nodes must decide in advance, how long they should be deactivated.

7.3 Isolated Approach

Based on the design decisions taken before, we now present our basic approach to schedule a single CN in a cluster. We call this approach *isolated* because it handles all CNs in a cluster as isolated entities. Possible relations between multiple CNs are ignored and only information concerning the single CN is used for scheduling decisions. An extended approach which takes into account information about all CNs in a cluster is presented in Section 7.4.

7.3.1 Deactivation Time

As discussed before, we use a fixed inactivity threshold strategy to determine when to deactivate a node. To do so, we introduce the inactivity threshold timeout Δt_{idle}. This timeout determines if a node has been idle long enough to be deactivated. When all services offered by a node become idle, i.e., no client uses any service of the node, a local timer is started. If a client contacts the node to use one of its services, the timer is stopped. After Δt_{idle} without any client contacting the node, the timeout occurs and the EnergyManager checks if the node can be deactivated. To do so, it uses the following information:

- *Application state*, i.e., if the applications and services executed on a node do currently allow the node to be deactivated. This may be not the case, if the node offers a service that is currently used or if it currently communicates with another node. This information is obtained from the BASE invocation broker.
- *Clustering state*, e.g., if a node is currently a CH or a CN in its probing phase. This information is used to identify that a given node may be inactive but cannot be deactivated due to SANDMAN itself, e.g., because it is currently working as a CH. The information about the clustering state is obtained from the cluster management.
- *Discovery state*, i.e., if a client has recently discovered a node or one of its services by sending a discovery request for it. If so, the node should stay active for a given time to give the possible client a chance to contact it. This information is obtained from the service management. Note, that this information is not available locally. It must be requested at the CH.

The EnergyManager decides to transition a node to its low power mode if and only if all services offered by the node are currently idle, all applications and services allow the node to be deactivated, the middleware reports that there are no interactions currently running, the cluster management reports that the node is currently a CN and not in its periodic probing phase, and the CH's service management confirms, that the node or one of its services have not been discovered by a client recently.

7.3.2 Deactivation Length

After deciding on the time of a node's deactivation, the energy management must determine, how long the node should be deactivated. To do so, the isolated approach uses the following information:

- The services offered by the CN and their *maximum acceptable usage delays*. This information is obtained from the service management.
- The remaining length of the CNs *clustering lease*. This information is obtained from the cluster management.
- The *minimum deactivation period* for the CN. This period gives the minimal length that the device must be deactivated in order to save energy. If the device is deactivated for a shorter period of time, the overhead for deactivating it is larger than the energy saved during deactivation. This period depends on the device at hand.

The maximum acceptable usage delay determines the upper bound for the length of a deactivation period. While a node is deactivated, it cannot be contacted by any client, forcing the client to wait for the node to awake. The maximal delay that clients are willing to wait depends on the kind of service they want to use. A client accessing a display may tolerate a lower delay than a client using a printer. We assume the service programmer to specify a fixed maximum usage delay when registering the service. In future work, this delay could be adapted dynamically due to a service's past usages or contracts with specific clients. Note, that this delay is experienced only for newly bound services, as a client may negotiate different client specific delays after first contacting the service. The maximum deactivation period T_1 resulting from the maximum usage delays of a given node v_i is given as:

$$T_1(v_i) := min(\{t_s^{max} | s \in S_{v_i}\}) \tag{7.1}$$

with S_{v_i} denoting the set of all services offered by the node v_i and t_s^{max} denoting the maximum acceptable usage delay for a service s (see Table 7.1).

Despite the maximum acceptable usage delay, the node may only be able to deactivate itself for a shorter delay, because of the length of its clustering lease. If the clustering lease expires while the node is deactivated, its CH may assume, that the node has been lost and may remove it from its cluster. Therefore, the node must be activated before the lease expires, to allow the cluster management to extend the lease. In SANDMAN, the energy management cooperates with the cluster management to achieve longer deactivation periods. To do so, the energy management notifies the cluster management that it intends to deactivate the node. The cluster management then tries to extend the lease immediately to maximize the possible deactivation time. We denote the remaining clustering lease length of a node v_i at time t as $L_{v_i}^t$.

Designator	Description
E_{act}	the energy consumed by transitioning a node from its deactivated to its activated state
E_{deact}	the energy consumed by transitioning a node from its activated to its deactivated state
Δt_{act}	the time needed to activate a node
Δt_{deact}	the time needed to deactivate a node
E_{anc}	the energy needed to send an announcement message
P_{sleep}	the power consumption of a deactivated node
P_{idle}	the power consumption of an activated idle node
S_{v_i}	the set of all services offered by the node v_i
t_s^{max}	the maximum acceptable usage delay for a service s

Table 7.1: Parameters used by the energy management (1)

Lastly, the minimal deactivation period must be chosen as the minimum sleeping time to save energy, the so-called break-even time (see [LCS+00], pp.2). This time depends on the values included in the power state machine of the individual node (see Figure 2.2), e.g., the power consumed to deactivate and activate a node. Using the definitions given in Table 7.1, the minimal deactivation period can be computed as:

$$T_{min} := (E_{act} + E_{deact} + E_{anc})/(P_{idle} - P_{sleep}) \tag{7.2}$$

Using all three components presented before, the deactivation period $T_{v_i}^t$ of a node v_i at time t is given as:

$$T_{v_i}^t := \begin{cases} (min(T_1, L_{v_i}^t)) & : \quad min(T_1, L_{v_i}^t) \geq T_{min} \\ 0 & : \quad otherwise \end{cases} \tag{7.3}$$

7.3.3 Deactivation Negotiation

As described before, after deciding to deactivate its node and computing the desired length of the deactivation, the local EnergyManager contacts the CH's EnergyManager to negotiate the actual deactivation. This is necessary due to a number of reasons. Most importantly, the CH's ServiceManager must be informed of the deactivation to update its service descriptions for this CN. This allows the CH to inform clients searching for services when the services offered by the deactivated CN can be used again. In addition, the CH may have new information about the current clustering state. As an example, the CH may have decided to forward the CH role to the requesting CN. Therefore, the CN is no longer idle and cannot be deactivated. Finally, the negotiation is used to perform some additional operations, e.g., renewing the cluster lease to maximize the deactivation length as discussed before. These operations are piggy-backed on the negotiation messages to save communication overhead.

The negotiation is done by sending the intended deactivation length to the CH. The CH checks this value and may decide to modify it due to its local information, e.g., a currently running cluster handover. After that, the CH returns a response message including the actual deactivation length. It may also deny deactivation at this time, e.g., to hand over its cluster to the requesting CN. After the CN receives the CH's response, it checks if deactivation is granted at this time. If so, it deactivates itself for the given length. After it has been reactivated, it restarts the process by initializing its local timer and waits for the next timeout Δt_{idle}.

An overview of the whole process is given in Figure 7.2. Here, at time t_0, CN v_i first starts

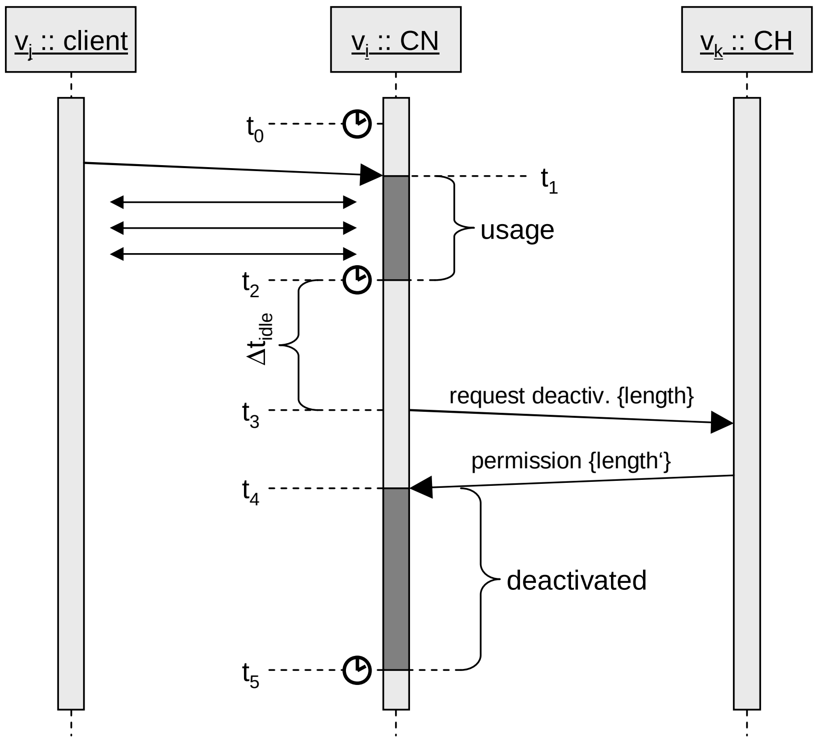


Figure 7.2: Isolated energy management example

its local timer and waits for client requests. At time t_1, client v_j contacts v_i and uses one of its services. The CN stops its timer and restarts it at time t_2, at which v_j stops using v_i. After Δt_{idle}, the timeout occurs and v_i checks if it can currently be deactivated (time t_3). As this is the case, it contacts its CH v_k and sends its intended deactivation length to it. The CH processes the message, updates its local service descriptions for v_i, and returns permission for deactivation to

v_i. Upon reception of this message (time t_4), v_i deactivates itself. After the predetermined time, it activates again, restarts its local timer and waits for client requests.

7.3.4 Transition Algorithm

After showing how the energy management decides that a node should be deactivated and how the deactivation period is computed, we define the complete algorithm used by the energy management. To do so, we need a special data structure, the IdleTimer, and two additional operations, `CalculateMinimalPeriod` and `CalculateDeactivationPeriod`, described below.

Data Structures and Operations

The central data structure of the isolated approach is the *IdleTimer*. It is used to decide if a node's services are currently unused and implements the inactivity threshold described in Section 7.2.1. To do so, IdleTimer interacts internally with the BASE Micro-broker (more specifically the ObjectRegistry) to be notified whenever a service is used (i.e., a request is received) and whenever a service becomes idle (i.e., a response is transmitted). IdleTimer includes a timer, that is initialized to the value of the inactivity threshold (Δt_{idle}) at startup time. Whenever the IdleTimer is notified that a service has received a request, it stops its timer. Whenever all services become idle, the timer is reset and restarted. If a timeout occurs, no service has been used for Δt_{idle}. In this case, IdleTimer fires a IdleTimerExpired event. IdleTimer offers three operations:

Start() The `Start` operation resets the internal timer to Δt_{idle} and starts the timer anew.

Stop() To stop the internal timer of the IdleTimer externally, the `Stop` operation is called.

IsExpired() returns Boolean `IsExpired` is used to check if the internal timer has expired, i.e., whether the timeout has already occurred. This can, e.g., be used to make sure that all services are still idle after an IdleTimerExpired event was catched.

In addition, the IdleTimer offers operations to subscribe for the IdleTimerExpired event, which are omitted here.

The `CalculateMinimalPeriod` operation calculates the minimal deactivation period T_{min} as defined in Equation 7.2. `CalculateDeactivationPeriod` computes the deactivation period as shown in Equation 7.3 with the exception, that it omits the current lease length from the calculation. This is the case, because the transition algorithm requests a new lease before initiating deactivation to maximize the achievable deactivation period. If `CalculateDeactivationPeriod` would include the current lease length in its calculation, it would request deactivation periods that are too short.

Transition Scheduling

At startup, before any service requests have been received, the EnergyManager subscribes for *IdleTimerExpired* events at the IdleTimer. After that, it starts the IdleTimer by calling the `Start` operation. While the system is executed, services offered by the node may receive requests, restarting the timer. If no service is used for Δt_{idle}, the timeout occurs and the OnIdleTimer-Expired operation at the EnergyManager is called. This operation is shown in Procedure 7.1.

Procedure 7.1 *OnIdleTimerExpired*: Called when the IdleTimer expires

```
onIdleTimerExpired()
1:  wait until:
2:      ($role$ = CN
3:      $\wedge$ not ClusterManager.IsProbing()
4:      $\wedge$ BASE.ObjectRegistry.AllIdle()
5:      $\wedge$ ClusterManager.IsIdle()
6:      $\wedge$ ServiceManager.IsIdle()
7:      $\wedge$ ServiceManager.ServiceRegistry.LastDiscovery(this)$\geq \Delta t_{idle}$)
8:      $\vee$ not IdleTimer.IsExpired();
9:  if IdleTimer.IsExpired() then
10:     Deactivate(); //initiate deactivation
11: else
12:     abort;
13: end if
```

It checks, if the node can be deactivated using the information discussed in Section 7.3.1. Note, that the operation contacts the local ServiceManager to determine when a service offered by this node has been discovered the last time. However, this information cannot be provided by the local ServiceManager. Instead, the remote ServiceManager of the node's CH must be contacted to determine this value correctly. However, this would require an additional request-response message exchange between the CN and its CH. To save communication, we follow a different approach. At this time, the ServiceManager checks locally for past discoveries. If none are found in the given time frame, it returns true and the node may decide to start negotiation the deactivation with its CH. The querying of the CH's ServiceManager about the real last discovery time is embedded into this negotiation process, saving two messages.

The time period that a service must not have been discovered in order to deactivate the node offering it is set to the inactivity threshold Δt_{idle}. This value is chosen, because Δt_{idle} denotes the time that the system grants a client to contact a service, either after the service provider has been activated or after the service has been discovered. Therefore, the system grants the same time to clients that discovered a service whose node is currently activated and client that discovered a service whose node is currently deactivated.

Initiating a Deactivation

After the EnergyManager has decided to deactivate the node as shown before in Procedure 7.1 it initiates deactivation by calling the `Deactivate` operation (see Procedure 7.2). `Deactivate` first checks how long the node can be deactivated by calling the `CalculateDeactivationPeriod` operation. If the deactivation period is larger than zero, the operation sends a SLEEP_ANC message to the CH. It includes the desired deactivation period and the device-dependent minimal deactivation period T_{min} (see Equation 7.2). As stated before, the EnergyManager cooperates with the cluster management to achieve longer deactivation times by renewing the cluster lease just before deactivation. To do so, the LeaseRequest message is piggy-backed on the SLEEP_ANC message. Therefore, before sending the SLEEP_ANC, the EnergyManager contacts the ClusterManager and requests a data package containing all data for the lease request. The returned data package is included into the SLEEP_ANC message and send with it. However, this is not sufficient. As discussed in Section 5.2.7, lease renewal requests are broadcasted in the CNs one-hop neighborhood, to enable other nodes to detect CNs reliably. As the SLEEP_ANC message includes a lease renewal, it must be broadcasted, too. Note, that we do not assume a reliable broadcast. Thus, the EnergyManager uses a timeout mechanism to detect if the CH answers in time. Otherwise, the EnergyManager assumes that the message was lost and resends it, until the cluster management detects that the CH is not available anymore and switches to the UN role.

Procedure 7.2 *Deactivate*: Deactivating a node

```
Deactivate()
 1: repeat
 2:   length := CalculateDeactivationPeriod();
 3:   if length > 0 then
 4:     data := ClusterManager.GetLeaseRequest();
 5:     minimal := CalculateMinimalPeriod();
 6:     bsend ({ch, EnergyManager}, SLEEP_ANC, {length, minimal, data});
 7:     start timer;
 8:     wait for message from ch;
 9:   end if
10: until answer received within timeout interval
```

Receiving a Deactivation Announcement

Upon reception of a SLEEP_ANC message, the *OnSleepAnc* operation (see Procedure 7.3) is called on the receiver. It first removes the set of data items containing the lease renewal information and hands it to the ClusterManager. The ClusterManager handles this as if a LeaseRequest message had arrived from the sender, with a small difference. Normally, the ClusterManager would return a GrantLease message containing the newly granted lease (see Section 5.6.1). In this case however, it grants the lease but does not send the message. Instead, it returns a new set of data items to the EnergyManager containing the data to send in the response message. Note, that if the receiver is not the sender's CH, the new data set is empty.

Procedure 7.3 *OnSleepAnc*: Receiving a SLEEP_ANC message

```
OnSleepAnc(sId, length, minimal, data)
1:  if ClusterManager.IsInCluster(sId) then
2:    retData := ClusterManager.SetLeaseRequest(data);
3:    if not ServiceManager.LastDiscovery(sId) $\geq \Delta t_{idle}$ then
4:      usend({sId, EnergyManager}, SLEEP_NACK, {retData});
5:    else
6:      lease := ClusterManager.GetLeaseLength(sId);
7:      length := min(length, lease);
8:      if length < minimal then
9:        usend({sId, EnergyManager}, SLEEP_NACK, {retData});
10:     else
11:       usend({sId, EnergyManager}, SLEEP_ACK, {length, retData}); ServiceManager.UpdateSchedule(sId, new Schedule(length));
12:     end if
13:   end if
14: end if
```

After this, the `OnSleepAnc` operation first checks if it is the CH of the sender. If this is the case, the operation checks if the deactivation can be allowed. To do so, it contacts the ServiceManager by calling the `LastDiscovery` operation with the sender's ID. If the service management reports that a client has recently discovered one of the sender's services, the EnergyManager denies deactivation and sends a SLEEP_NACK message to the sender, including the cluster lease related data package received from the ClusterManager. Otherwise, the EnergyManager checks if the requested deactivation period can be granted. To do so, it requests the remaining lease duration of the sender from the ClusterManager. Note, that this is the lease duration after the ClusterManager has already processed the lease renewal request and may have granted a new lease. In addition, the ClusterManager automatically removes a portion of the actual lease length before returning it to the EnergyManager. This gives the node receiving the lease enough time to renew its lease after its reactivation. The granted deactivation period is the minimum of the remaining lease length and the requested deactivation period.

In certain cases, the cluster management will grant only a minimal lease, e.g., if the CH is currently handing over its cluster to another node. In such cases, the achievable deactivation period will be very short. Before sending the deactivation period, the EnergyManager compares it with the minimal deactivation period. If it is smaller, the EnergyManager denies deactivation and sends a SLEEP_NACK message as discussed before. Otherwise, the EnergyManager includes the granted deactivation period into a SLEEP_ACK message and sends the message back to the sender. Again, it includes the data returned from the cluster management earlier into the message. Finally, the EnergyManager notifies the local ServiceManager about the newly granted deactivation period using the `UpdateSchedule` operation (see Section 6.7.2).

The reason for the CH checking whether the deactivation period is longer than T_{min} instead of letting the CN do so when receiving the SLEEP_ACK message from the CH is that in the latter case the CH would wrongly expect the CN to be deactivated. Its ServiceManager would report this fact to clients when answering lookup requests, letting them wait until the false deactivation period exceeds.

Deactivation Denial

If a SLEEP_NACK message is received, the EnergyManager extracts the lease related data from the message and hands it to the local ClusterManager. After that, it restarts the IdleTimer.

Deactivation Permission

In case a SLEEP_ACK message is received, the `OnSleepAck` operation (see Procedure 7.4) is called. Analogous to `OnSleepNAck`, this operation first extracts the data package containing the cluster lease renewal and hands it to the ClusterManager. After that, `OnSleepAck` rechecks, if the local node is still able to be deactivated. This could not be the case, because the node could have received a service request in the meantime. It could also be the case, because the cluster management could have started a new periodic probing phase when granting the new lease.

Note, that if the deactivation check succeeds, the granted deactivation period must be adapted before deactivation can start. First, the time between originally computing the favored time and receiving the permission must be accounted for. We denote this time as *reqRespDelay*. To measure it, the CN starts a timer when sending the original SLEEP_ANC message and stops it when the SLEEP_ACK is received. The EnergyManager subtracts this time from the granted period (see Procedure 7.4, line 3). In addition, both deactivation and activation may not be instantaneous but may take some time, the so-called deactivation and activation delays. The

Procedure 7.4 *OnSleepAckReceived*: Receiving a SLEEP_ACK message

```
OnSleepAckReceived(length, data)
 1: ClusterManager.SetLeaseRequest(data);
 2: if $role$ = CN
        $\wedge$ not ClusterManager.IsProbing()
        $\wedge$ BASE.ObjectRegistry.AllIdle()
        $\wedge$ ClusterManager.IsIdle()
        $\wedge$ ServiceManager.IsIdle()
        $\wedge$ IdleTimer.IsExpired()
     then
 3:   length := length - reqRespDelay;
 4:   length := length - $\Delta t_{act}$;
 5:   if length > $\Delta t_{deact}$ then
 6:     ExecuteDeactivation(length);
 7:   else
 8:     IdleTimer.Start();
 9:   end if
10: else
11:   IdleTimer.Start();
12: end if
```

EnergyManager subtracts the activation delay from the granted deactivation period to make sure, that the node's reactivation starts early enough (see Procedure 7.4, line 4). Finally, it must check if the remaining deactivation period is larger than the deactivation delay Δt_{deact}, allowing the node to be deactivated before it should wake up again. If the check is successful, deactivation is finally executed by calling the ExecuteDeactivation operation. Otherwise, the node must stay active and the IdleTimer is restarted.

Clearly, this leads to the CH temporarily distributing wrong information about the node, as the CH thinks that it is deactivated. To correct this, the CN could contact its CH again, notifying it that it is still active. However, this case can only happen, if the deactivation period granted by the CH was slightly larger than T_{min}. In general, the additional energy consumption for sending a message to the CH could be higher than the energy consumed by a client falsely waiting for the node to awaken. There may be no such client at all. Therefore, we omit sending this notification to the CH. In reality this case is most unlikely as the only reason for a very small deactivation period is a cluster modification underway by the CH. In such cases, the minimal lease granted to the CN normally leads to the CH denying deactivation in the first hand.

Deactivation and Reactivation

After the final deactivation period is computed, the node contacts all system parts and notifies them of the upcoming deactivation. This allows the system to reach a stable state before deactivation is started. In our system model, we assume that a node can be deactivated without loosing its state. Therefore, in our system the system must not perform any complex operations here. Without this assumption the system would e.g., need to save its state to stable storage.

When all system parts have successfully prepared themselves for deactivation, the EnergyManager initiates the local watchdog timer with the granted deactivation period and deactivates the device. After the given time, the watchdog timer activates the device again and the ServiceManager restarts the IdleTimer.

7.3.5 Discussion

The isolated approach is rather straight forward. It allows a CN to detect that it is currently not needed and to deactivate itself after notifying the CH of its intention. The CN's deactivation period is calculated depending on the remaining lease length and the maximum acceptable usage delays for all services offered by the CN. For long acceptable usage delays, this results in long deactivations and therefore large energy savings. If the usage delay must be short, however, this approach cannot achieve large savings. Using the minimal deactivation period, it avoids spending energy unnecessarily if the deactivation periods are too short to be beneficial. In such cases, the CN will stay active continuously.

7.4 Coordinated Approach

The basic isolated approach is able to deactivate nodes depending on the maximum usage delays acceptable for services offered by the node. If high usage delays can be tolerated, the isolated approach can deactivate nodes for long periods of time. However, if a service must be accessible with little delay, the isolated approach may not be able to achieve long deactivation periods or may need to keep a device active altogether. In this section, we extend the isolated approach such that not only information about a single node but about all nodes in the cluster are taken into account when scheduling the node's deactivation times.

The coordinated approach resembles the isolated approach closely. It uses the same data structures, messages and timers as the isolated approach. The main difference between both approaches is the computation of the deactivation period.

The main idea of the coordinated approach is that if a client v_i needs one service matching a certain description d, it can often choose between several services matching d. In this case, the usage delay experienced by v_i is the minimum of the usage delays of all these services, as v_i will choose the service which can be used the soonest. Therefore, to minimize the usage delay experienced by v_i we have to spread the deactivation periods of all nodes offering a matching service evenly over time. This allows v_i to always pick a service that will be available shortly and thus allows the nodes to be deactivated longer.

7.4.1 Transition Scheduling

In the coordinated approach, the deactivation of a CN is scheduled exactly as in the isolated approach (see Section 7.3.1). Again, the CN uses its application state, clustering state and discovery state to decide when to initiate its deactivation.

However, the deactivation length is computed differently. Informally, our approach is based on the CH memorizing for each service type offered in its cluster the next time that a service of this type will be available. When a CN wants to be deactivated, the CH looks up this time for each service offered by the CN. By adding the maximum acceptable usage delays for the service types, the CH can compute when the CN must be reactivated in order to offer its services in time. As an example, take a cluster that contains two nodes v_1 and v_2, both offering a service (s_1 and s_2) of the same service type. We assume that the maximum acceptable usage delay t_s^{max} for these services is given as 10 s. At a given time, the CH has scheduled v_1 to awake in 8 s, when v_2 contacts it to be deactivated. The CH knows that s_1 will be available in 8 s. Therefore, it schedules v_2 to awake in $8s + t_s^{max} = 18s$. This heuristic approach can be computed very efficiently.

More formally, given the definitions depicted in Table 7.2, the CH computes the granted deactivation time as:

$$T_{v_i}^t := \min(\{(\Delta t_s^{next} + \Delta t_s^{max}) | s \in S_{v_i})\} \bigcup \Delta t_{v_i}^{lease}) \tag{7.4}$$

Designator	Description
S_{v_i}	the set of all services offered by the node v_i.
Δt_s^{max}	the maximum acceptable usage delay for a service s.
Δt_s^{next}	the next time that a node offering a service of the same type as s will be available in the cluster.
$\Delta t_{v_i}^{lease}$	the current lease length for node v_i.

Table 7.2: Parameters used by the energy management (2)

An example of the resulting system behavior can be seen in Figure 7.3. The system contains

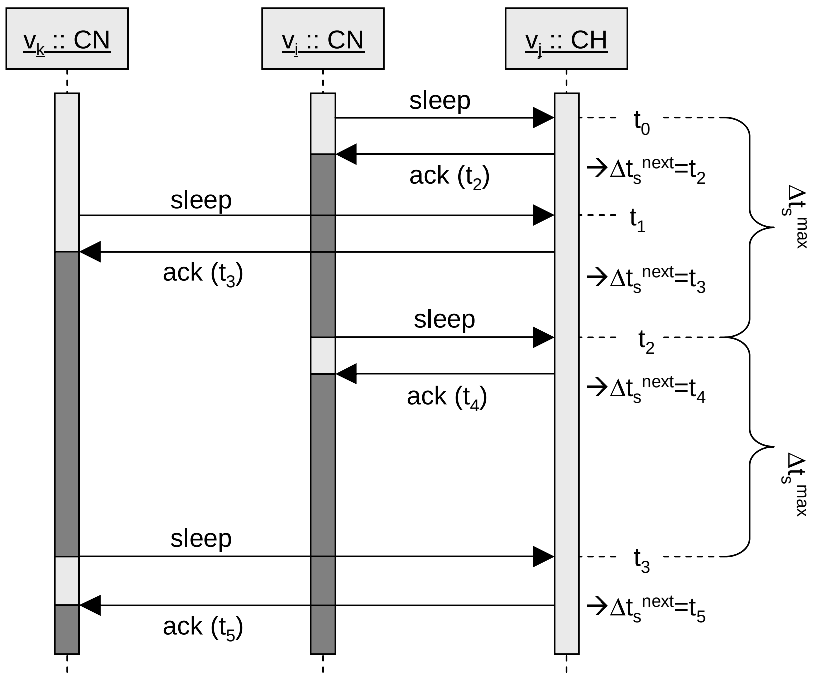


Figure 7.3: Coordinated energy management example

of three nodes, v_i, v_j and v_k. Node v_k is a CH, v_i and v_j are members of its cluster. Both offer the same service s. At time t_0, v_j decides to deactivate itself and sends a sleep announcement message to the CH v_k. The CH checks the future availability of s (Δt_s^{next}) and finds, that it does not have any information about this, yet. Therefore, it adds the maximum acceptable usage delay for s (Δt_s^{max}) to the current time t_0, leading to $t_2 = t_0 + \Delta t_s^{max}$. It returns t_2 to the requesting node and sets $\Delta t_s^{next} = t_2$. A little later, at time t_1, v_i requests to deactivate itself, too. Again, the CH checks, if it has any information about when v_i's service s will be available in the future. This time, it finds that s will be available again at t_2. Thus, it adds the maximum delay to t_2, leading to $\Delta t_s^{next} = t_3 = t_2 + \Delta t_s^{max}$ and deactivates v_i until this time. As we can see, this approach leads to s being available after at most Δt_s^{max}, although v_i (and v_j after the first round) can be deactivated for a longer time than Δt_s^{max}, as the system knows that there is already another node that will provide the service at a future point of time. Therefore, it can schedule the new node behind this node without violating the maximum usage delay. This example can be extended for multiple nodes with different sets of services.

In the following we extend this idea into an algorithm that aims at spreading the deactivation periods of several nodes that offer the same types of services evenly over time.

7.4.2 Transition Algorithm

The transition algorithm used by the coordinated approach resembles the one of the isolated approach closely. It contains two differences that are described in this section, namely in the `Deactivate` and the `OnSleepAnc` operations. All other operations are the same as the ones used in the isolated approach and are omitted in this section.

Additional Data Structures

In addition to the data structures and operations used by the isolated approach, the coordinated approach uses another data structure, the **SchedulingTable**. Each CH maintains its own SchedulingTable locally. The SchedulingTable includes an entry for each service type in the cluster. Each entry e contains the service type $e.type$, and the most current future activation time of a service of this type $e.time$. As an example, if a given service type will be active in 10, 20 and 30 sec, its entry will contain 30. A new entry is initialized with $e.time := 0$. To include a new entry in the table or to update an existing entry, the *setFutureTime* method can be called with the service type and the future time as parameters. To retrieve the future time for a given service type, the *getFutureTime* method can be called with the service type as parameter.

Modifications to Initiating a Deactivation

As noted before, the coordinated approach differs from the isolated approach in two main points. The first difference is located in the way a deactivation is initiated. As in the isolated approach, this is done by sending a SLEEP_ANC message to the CH. However, in the coordinated approach the CN does not include an intended sleep length into the message. Instead, the calculation of this length is done entirely by the CH. Therefore, the `Deactivate` operation is changed slightly, as shown in Procedure 7.5.

Procedure 7.5 *Deactivate*: Deactivation of a node

```
Deactivate()
1: repeat
2:     $data$ := ClusterManager.GetLeaseRequest();
3:     $minimal$ := CalculateMinimalPeriod();
4:     bsend ({$ch$, EnergyManager}, SLEEP_ANC, {$minimal$, $data$});
5:     start timer;
6:     wait for message from $ch$;
7: until answer received within timeout interval
```

Modifications to Receiving a Deactivation Announcement

The second difference between the algorithms is the handling of a SLEEP_ANC message. Once a CN has decided to deactivate itself, it sends a SLEEP_ANC message to its CH, containing its intended deactivation period. In the isolated approach, the CH simply accepted the deactivation period proposed by the CN, if it felt, that it was save to grant it. Otherwise, it shortened the period accordingly or forbid deactivation altogether. In the extended approach, the CH computes the deactivation period following Equation 7.4. The new `OnSleepAnc` operation is shown in Procedure 7.6. The new computation of the deactivation period is contained in line 12 to 18.

Procedure 7.6 *OnSleepAnc*: Receiving a SLEEP_ANC message

```
OnSleepAnc($sId$, length, minimal, data)
 1: if ClusterManager.IsInCluster($sId$) then
 2:   retData := ClusterManager.SetLeaseRequest(data);
 3:   Services := ServiceManager.GetServices($nId$);
 4:   for all s $\in$ Services do
 5:     if SchedulingTable.Get(s.$type$) < $t_{now}$ then
 6:       SchedulingTable.Set(s.$type$, $t_{now}$);
 7:     end if
 8:   end for
 9:   if notServiceManager.LastDiscovery($sId$)$\geq \Delta t_{idle}$ then
10:     usend({$sId$, EnergyManager}, SLEEP_NACK, {retData});
11:   else
12:     length := ClusterManager.GetLeaseLength($sId$);
13:     for all s $\in$ Services do
14:       length := min(length, SchedulingTable.Get(s.$type$)+$\Delta t_s^{max}$));
15:     end for
16:     for all s $\in$ Services do
17:       SchedulingTable.Set(s.$type$, length);
18:     end for
19:     if length < minimal then
20:       usend({$sId$, EnergyManager}, SLEEP_NACK, {retData});
21:     else
22:       usend({$sId$, EnergyManager}, SLEEP_ACK, {length, retData}); ServiceManager.UpdateSchedule($sId$, new Schedule(length));
23:     end if
24:   end if
25: end if
```

7.5 Conclusion

The energy management detects idle nodes and deactivates them if possible. SANDMAN can integrate different scheduling algorithms for deactivating nodes and determining their deactivation periods. In this chapter we presented two algorithms for scheduling node deactivation periods, an isolated algorithm, scheduling each member of a cluster individually, and a coordinated algorithm, which interleaves the deactivation periods of CNs offering the same services in order to achieve lower usage delays.

8 Evaluation

In this chapter, we evaluate the performance of our service discovery system SANDMAN in different scenarios. Our evaluation is structured as follows: in Section 8.1 we present the performance metrics used to rate our approach. In Section 8.2 we discuss the system parameters and how they influence our performance metrics. After that we derive an analytical model that allows to assess and configure SANDMAN in different scenarios. In order to capture effects induced by mobility we additionally conducted experiments using an emulation environment. These experiments are presented after the analytical model. Finally, we discuss the achieved results. We end the chapter with a short summary and conclusion in Section 8.5.

8.1 Performance Metrics

To quantify the performance of the proposed algorithms we measure a number of parameters. The main goal of SANDMAN is to provide an energy efficient service discovery. Therefore, the achieved energy savings are an important performance metric. In addition, in Section 4.1 we have specified a number of objectives, which a discovery system should fulfill, namely discovery latency, precision and recall. These objectives characterize different aspects of a successful discovery request and should be evaluated, too. Finally, we measure the message overhead of our approach to quantify how much of the communication channel is used up by SANDMAN. In the following we describe these performance metrics in more detail.

Energy Savings

SANDMAN saves energy by temporarily deactivating devices. To do so, it introduces a certain communication overhead needed to execute the cluster and energy management. The achievable energy savings can be calculated as the difference between the energy saved by deactivating nodes and the energy spend to execute SANDMAN. We provide energy savings as percental values. This allows to correlate the savings with the total consumption.

Discovery Latency

The discovery latency is defined in Definition 4.1 on page 64 as the time difference between a client starting a discovery request and the answer being reported back to this client. It is important to achieve a low discovery latency in order to provide clients with new information about services as fast as possible, e.g., to plan application adaptations by switching to other services. This latency depends heavily on the kind of discovery request placed by the client as discussed in Section 6.5. It is given in milliseconds.

Discovery Precision

The discovery precision is given in Definition 4.2 on page 64. As described there, it gives the fraction of correctly discovered services compared to the total number of discovered services. It is provided as a percentual value.

Discovery Recall

The discovery recall describes how many of the services that should be discovered are really discovered. It is defined in Definition 4.3 on page 65. Identically to the discovery precision, the discovery recall is given as a percentual value.

Message Overhead

Lastly, we measure the message overhead caused by SANDMAN. This overhead consists of all messages additionally send due to the usage of our system in comparison to using a classical peer-based approach without any continuous operations. As an example, messages send to form a new cluster are specific to SANDMAN and contribute to the message overhead. Messages send to find a specific service on the other hand are not specific to our approach, as a classical approach would send them, too. Therefore, these messages are not part of the message overhead. Measuring the message overhead allows to characterize the influence of our approach on the communication channel.

8.2 System Parameters

With the performance metrics given in Section 8.1 in mind, we can now analyze the influence of different system parameters on the system's performance. In this section, we introduce the relevant system parameters and describe their influences. We group the system parameters in four groups: the mobility model parameters, the energy model parameters, the environmental system parameters, and the SANDMAN parameters, which we discuss in turn.

Mobility Model

One of the most important influence factors in our system is the node mobility model. In contrast to other systems, it is insufficient to model node mobility as a random waypoint movement. Instead, as SANDMAN is designed for node groups with similar movement patterns, the mobility model should offer group mobility. Different group mobility models have been designed and published. For our evaluation, we use the well-known Reference Point Group Mobility (RPGM) model [HGPC99]. In RPGM each group has a center, around which all group members are positioned. By moving the center, the whole group moves together but an additional micro-movement can be defined for each group member. The center itself is moved following a random waypoint pattern.

The RPGM model used by us offers six parameters, as shown in Table 8.1: the average group speed and pause time define the random waypoint movement of the group center. The group size and the standard group size deviation define the number of nodes per group and how homogeneous the groups are in size. The distance to the group center specifies the maximum distance that a node in a group should have to the group center, and the group switch probability gives the probability that a node may switch to another mobility group when it is near this group. Note, that a group may be empty in this model, i.e., all nodes of a group may switch to other groups. At a later point of time, another node may enter the group, again.

The mobility model influences all performance metrics. If nodes move randomly, i.e., with a group size of 1, and with relatively high movement speed, e.g., 10 m/s, they will rarely stay together long enough to be clustered, thus no energy will be saved. As another example, if nodes move faster, topology changes will typically become more frequent, leading to lower discovery precision and recall.

Due to their importance, the average group speed v_{grp} and the group size n_{grp} should be analyzed in more detail. Group speed directly influences the level of node mobility experienced. For different node mobilities, the system will behave quite differently. Lower mobility should lead to more stable clusters and thus higher savings and better precision and recall. Group size influences how many CNs a CH normally has. As one extreme, all nodes in the system may

Parameter	Description
v_{grp}	The average speed of the group center.
Δt_{grp}	The average time that a group will stay at a given destination after reaching it. After this time, the group will pick another destination and start to move there.
n_{grp}	The number of nodes in a group.
σ_{grp}	The standard deviation from the group size given before.
$dist_{grp}$	The maximum distance of a node to the center of its group.
p_{grp}	The propability that a node will switch to another group once it comes near it.

Table 8.1: Mobility model parameters

form a single mobility group, leading to a mostly static system without mobility. Here, long clustering times should be common. As another extreme, if each group has only one member, the RPGM model is equivalent to a random waypoint model. In such a scenario, shorter clustering times, or – depending on the group speed – no clustering at all may arise.

Higher average group pause times Δt_{grp} lead to node groups staying at fixed positions for longer periods of time before moving to another position. If nearby groups pause, they may be clustered, when – without pauses – they would not have been. In general, longer pauses lead to more stable systems, with less mobility.

The distance to the group center $dist_{grp}$ influences how far away the mobility trace generator places nodes of the same mobility group. This influences the probability of nodes moving together having different neighborhoods. In such cases, no clustering should occur or an already existing clustering will be dissolved. Otherwise, discovery precision and delay could suffer, as clients may miss service provider nodes or CHs may announce services that are unreachable for a client. Thus, by increasing the distance, we can increase the influence of our approach on discovery precision and recall. The upper limit for the distance to the group center is half of the communication range. Otherwise, nodes that are in the same mobility group may not be able to communicate with each other. Therefore, they are not clustered.

Using the standard group size deviation σ_{grp} we can influence whether all groups start with the same amount of nodes, thus allowing all clusters having the same number of CNs. By increasing this value, we can produce more different group sizes. In our evaluation, we want to control the group size in order to control its influence on our results. Therefore, to maximize control, we use a standard group size deviation of zero and vary the group size instead.

The group switch probability p_{grp} can be used to influence the group stability. With a switch probability of zero, we have completely stable groups. With higher probabilities, nodes will change between groups more often. Note, however, that the probability is computed continuously. Therefore, while two groups are nearby, a node may switch multiple times. This can lead to a node seemingly staying in the same group even with a switch probability of one, because the node oscillated between the groups continuously until both groups leave each others influence area.

Energy Model

Besides the mobility model, the node's energy model is a very important factor. We have defined our energy model for nodes in Section 2.4. The most important parameters of this model are given in Table 8.2.

Parameter	**Description**
Δt_{act}	The time needed to activate a node.
Δt_{deact}	The time needed to deactivate a node.
P_{send}	The power consumed by a node to send data for one second.
P_{recv}	The power consumed by a node to receive data for one second.
P_{idle}	The power consumed by a node for idle standby.
P_{sleep}	The power consumed by a node while being deactivated.
E_{deact}	The energy needed to deactivate a node.
E_{act}	The energy needed to activate a node.

Table 8.2: Energy model parameters

The achievable energy savings depend largely on the parameters of the energy model. Most importantly, the bigger the difference between P_{sleep} and P_{idle}, the more energy can be saved by temporarily deactivating nodes, as the maximum achievable saving per second is defined as:

$$P_{max} := P_{idle} - P_{sleep} \tag{8.1}$$

On the other hand, the energy consumption introduced by the message overhead of our system depends on the difference between P_{send} and P_{idle} as well as the difference between P_{recv} and P_{idle}. With higher differences, sending and receiving messages becomes increasingly costly compared to simply waiting for messages to arrive.

The delay and energy consumption for switching between activated and deactivated mode influences energy savings, too. If switching between energy modes is very costly, each switching operation should be considered carefully. In addition, the switching delay gives a lower limit to the deactivation length Δt_s (see Table 8.4 on page 158):

$$\Delta t_s \geq \Delta t_{act} + \Delta t_{deact} \tag{8.2}$$

Otherwise, the node could not be reactivated in time.

Environmental Parameters

The environmental parameters determine the environment of our system, e.g., its size or the number of nodes present in the environment. An overview of the environmental parameters is given in Table 8.3.

Parameter	**Description**
w	The width of the experimental area in meters.
h	The height of the experimental area in meters.
r_{tx}	The transmission range of the used communication technology in meters.
d_{tx}	The data rate of the used communication technology in bytes per second.
n	The total number of nodes in the system.
S_{avg}	The average number of services offered by a node.
D_{avg}	The average size of a description record of a service in byte.
sizeof(m)	The size of a message of the given type m in byte.
$frac_{idle}$	The average percentage of the nodes' life times during which they are idle.

Table 8.3: Environmental parameters

The area width and height, in combination with the transmission range and the number of nodes define the node density of a scenario. The density influences the number of neighbors, and thus the number of service providers that could be used by a client. In addition, the number of neighbors can influence the energy savings. If the nodes cannot be clustered, a lot of announcement (ANC) messages are received by each node, each consuming energy. If they can be clustered, the cluster size will increase and more energy may be saved.

In addition, the transmission range, together with the nodes' movement speed influences the duration that two nodes stay in communication range. With higher ranges, nodes can communicate longer. Therefore, longer deactivation periods are achievable and thus higher energy savings are possible. Note, however, that a higher communication range can also result in nodes having different neighbors, as far away nodes may become visible to them. In such cases, the nodes could not be clustered and thus could not save energy. In our evaluation we keep the transmission range fixed and vary the movement speed instead.

The communication data rate combined with the transmission powers P_{send} and P_{recv} determines the energy spent to transmit a message of a given size. Therefore, with a higher data rate, more data can be send with the same energy overhead. Note that a given communication technology might use different P_{send} and P_{recv} for different data rates, possibly countering this effect.

We denote the size of a message of type m as *sizeof(m)*. As an example, *sizeof*(ANC) gives the size of an announcement message in byte. Larger message sizes lead to more data traffic and consumes more energy. The size of the messages depend largely on the BASE plugin used to marshal the message. As this plugin may change in BASE dynamically, message sizes may vary at runtime. For the sake of simplicity, we assume fixed message sizes for our evaluation.

Each node may offer a number of services. In line with Table 4.1, we define S_{v_i} as the set of all services offered by a node v_i. Hence, the average number of services per node can be computed as:

$$S_{avg} := \frac{\sum_{i:=1}^{n} |S_{v_i}|}{n} \tag{8.3}$$

The average number of services per node and the average size of a service description record influence the size of join and handover messages. If these messages are too large, they may undo our energy savings. However, for this to happen, we would need a huge number of services per node. Using Equation 3.3 on Page 51 we can compute that if we deactivate a node for one second, enough energy is saved to transmit a message containing about 304 service descriptions, with an average descriptor size of 512 byte. Thus, to counter the energy savings of a cluster with 10 CNs, each being deactivated for 10 s, a handover message has to contain about 30400 services.

The idle percentage $frac_{idle}$ influences the energy savings and usage latency. If a node is seldomly idle, it cannot be deactivated for most of the time, thus saving no energy. On the other hand, it is readily available for most of the time, resulting in a low usage latency.

SANDMAN parameters

The fourth group of system parameters contains all parameters that can be used to control our service discovery system SANDMAN. An overview of these parameters is given in Table 8.4.

Parameter	Description
γ	lifetime comparison threshold
Δt_s	average deactivation time
Δt_i	idle timeout
$lease_{avg}$	average lease length
$lease_{max}$	maximum lease length
f_{anc}	announcement frequency
f_{prob}	probing frequency
Δt_{prob}	probing length
$lease_{tol}$	The safety tolerance used by a CN to extend its lease at the CH.
Δt_{join}	clustering delay

Table 8.4: SANDMAN parameters

The threshold value γ is used when a node checks whether it should join another cluster (see Equation 5.1) to determine whether the lifetimes of two nodes are equal or not. By changing γ, we can modify the frequency of CH changes. If γ is small, small lifetime differences will lead to the CH handing over its role to another node. Thus, the system is very fair, as the overhead for being CH is shared equally between the nodes. However, each CH switch consumes energy. To save additional energy, larger γ can be used. The exact value depends on the given scenario and its requirements concerning fairness. If all devices belong to one user, the user might favor higher energy savings and thus fewer CH changes. If the devices belong to multiple users, they might prefer the overhead being shared fairly between their devices.

The deactivation period Δt_s is one of the most important parameters for tuning SANDMAN's behavior. It controls, how long nodes are deactivated and thus directly influences the achievable energy savings. In addition, Δt_s influences the discovery precision and recall. If nodes are deactivated for longer periods of time, new or lost nodes are detected later, possibly leading to lower precision and lower recall. To balance the energy savings with precision and recall, the length of the deactivation period Δt_s must be chosen according to the current level of mobility. The exact influence of Δt_s is evaluated both in our analytical and experimental evaluations.

The idle timeout Δt_i makes sure, that all clients waiting for a given service are able to contact the service after its provider node was reactivated. Its length depends on several factors. On the one hand, it should be as short as possible to let idle nodes deactivate themselves sooner.

On the other hand, if it is too short, clients will miss the contact window and will have to wait for another deactivation period. This leads to higher usage delays. Clients may miss the window due to communication delays. Therefore, the window size depends on the typical communication jitter. This jitter is bigger for nodes that are far away and must therefore send their messages over many hops. It also depends on the MAC protocol used, as clients may not be able to send messages because other nodes communicate currently.

As already discussed in the energy management, lease lengths depend on Δt_s, Δt_i and $lease_{tol}$. To achieve a given deactivation length, the length of leases must be larger than the sum of these parameters. Otherwise the CH could remove deactivated CNs from its cluster or additional lease requests could be necessary.

$$lease_{avg} \geq \Delta t_s + \Delta t_i + lease_{tol} \tag{8.4}$$

The probing length Δt_{prob} depends on the maximum lease length $lease_{max}$, as the probing must last long enough to make sure that at least one message of each currently present node is received. As the messages received by the probing from CNs are lease requests, the probing must therefore last at least as long as the maximum lease length. Therefore, in order to not waste energy by using too long probings, the maximum lease length should be set to the minimal possible value given by Equation 8.4.

The clustering delay Δt_{join} determines, how long two nodes must be in communication range before they are clustered. Thus, Δt_{join} can be used to tailor how conservative the system is in creating clusters. If Δt_{join} is small, nodes are clustered more often, saving more energy. However, if the clustering was too optimistic and clustered nodes are lost after a short time, discovery precision and recall may suffer greatly. On the other hand, for a large Δt_{join}, the system will react slowly to newly appearing nodes, hence saving less energy.

Announcements are sent regularly using the announcement frequency f_{anc}. This parameter influences the message overhead, as with higher values, messages are send more often. Clearly, this also influences the energy savings, as more energy must be spend to send announcements. Note that f_{anc} indirectly also influences how fast nodes are clustered. The reason for this is, that a new neighboring UN or CH is detected with its first ANC message. Therefore, assuming a normal distribution of the arrival times of new nodes, the average delay between the node arriving in communication range and other nodes detecting it can be modeled as $\frac{1}{2*f_{anc}}$. Once a node is detected, the clustering delay Δt_{join} is started. To determine, that the node is still present, other nodes listen to its ANC messages. When, Δt_{join} after first detecting the new node, the node sends its next ANC message, the others may detect, that the node can be clustered. Depending on f_{anc} and Δt_{join}, this ANC message may be send anytime between Δt_{join} and $\Delta t_{join} + \frac{1}{f_{anc}}$, thus again leading an average additional delay of $\frac{1}{2*f_{anc}}$. Therefore, the average delay Δt_{avg} before clustering a newly arrived node is:

$$\Delta t_{avg} := \frac{1}{2 * f_{anc}} + \Delta t_{join} + \frac{1}{2 * f_{anc}} = \Delta t_{join} + \frac{1}{f_{anc}} \tag{8.5}$$

8.3 Analytical Evaluation

Due to the high complexity of the SANDMAN system, a complete analytical evaluation of its behavior is practically impossible. Nevertheless, in order to deal with this high complexity, we develop a number of analytical models, each focusing on one specific performance metric and the parameters influencing it. These models serve two purposes. First, they can be used to derive concrete parameter settings, e.g., deactivation durations, for given scenarios and assess their trade-off, for instance energy savings versus latency. This is further discussed in Section 8.3.5. Second, using this information we can identify scenarios for which a more detailed evaluation using experiments should be performed. As discussed in Section 8.3.2 this is foremost the case for the stability of the system, since this parameter influences the achievable savings to a large degree. Therefore, we performed a number of experiments with different levels of stability in Section 8.4.

In our analytical evaluation, we assume the following simplifications: first, we omit service providers offering new services at runtime or stopping to offer services at any time. Therefore, no Register, Update or Remove messages are sent. These events are rare and will not influence our system performance significantly but would complicate our analytical models further. Second, a CH keeps its role for a given time, before handing it over to another member of its cluster. In reality, we use the threshold value γ to determine, that a node has a sufficiently larger remaining lifetime than the current CH. We then hand over the cluster to this node. In the analytical evaluation we omit γ and use a fixed duration for staying CH instead. We call this duration Δt_{ch}.

8.3.1 Message Overhead

The first performance metric that we want to analyze is the message overhead of our system. In SANDMAN, messages are sent to create and maintain clusters, to deactivate nodes, and to distribute discovery lookup requests and subsequent service announcements. In the following, we first analyze the message overhead resulting from clustering and deactivation. After that, we discuss the overhead of finding services.

Clustering and Deactivation

Clustering and deactivation messages form the basic message overhead. They are sent during the whole system life time and are independent of the actual usage of the system to find services.

To join a cluster, two messages are needed, a JOIN_REQ and a JOIN_ACK message. As shown in Section 6.6.1 the registration of all services of the joining node is done implicitly in the JOIN_REQ message and no additional messages are needed at this point. To form a cluster with n nodes, this join protocol must be performed for all but the CH, i.e., $(n-1)$ times. Therefore, the message overhead MO_{join} to form a cluster with n nodes is:

$$MO_{join} := 2 * (n-1) \tag{8.6}$$

In addition, UNs and CHs emit ANC messages regularly. Depending on the announcement frequency f_{anc}, the message overhead per second for a network with n unclustered nodes is:

$$MO_{anc} := n * f_{anc} \tag{8.7}$$

Using Equations 8.6 and 8.7, we can specify the message overhead MO_{init} for a system starting with n unclustered nodes and forming a single cluster with an average clustering delay Δt_{avg} (see Equation 8.5) as:

$$MO_{init} := \Delta t_{avg} * MO_{anc} + MO_{join} = \Delta t_{avg} * n * f_{anc} + 2 * (n-1) \tag{8.8}$$

The resulting overhead is shown in Figure 8.1 for $\Delta t_{join} = 120s$ and different numbers of nodes n and announcement frequencies f_{anc}. Note that altering Δt_{join} instead of f_{anc} leads to similar results as Δt_{join} essentially influences how long ANC messages are sent. Changing f_{anc} controls how often ANC messages are send. Therefore, prolonging the time to send such messages is equivalent to sending them with a higher frequency in terms of message overhead.

Once clustered, the message overhead per second of a single cluster with n idle nodes is characterized by the CH sending ANC messages regularly and the CNs requesting their deactivation periods. In addition, the CH forwards its role regularly. For this analytical model, we use a simplified constant time factor Δt_{ch} denoting how long a CH performs its role. In a real system this value would vary depending on the actual energy usage of the nodes in the cluster. The resulting formula is:

$$MO_{cont} := f_{anc} + \frac{(n-1)*2}{\Delta t_s + \Delta t_i} + \frac{2}{\Delta t_{ch}} \tag{8.9}$$

Using this formula, we can compute the message overhead of a running cluster as shown in Figure 8.2 for $\Delta t_{ch} = 300s$ and $f_{anc} = \frac{1}{30s}$ We vary the number of nodes n and the sum $(\Delta t_s + \Delta t_i)$. The single values of Δt_s and Δt_i are unimportant for the amount of messages sent. As can be easily seen in the formula, Δt_{ch} and f_{anc} influence the message overhead less than $(\Delta t_s + \Delta t_i)$, as only two messages are sent per handover and one per announcement.

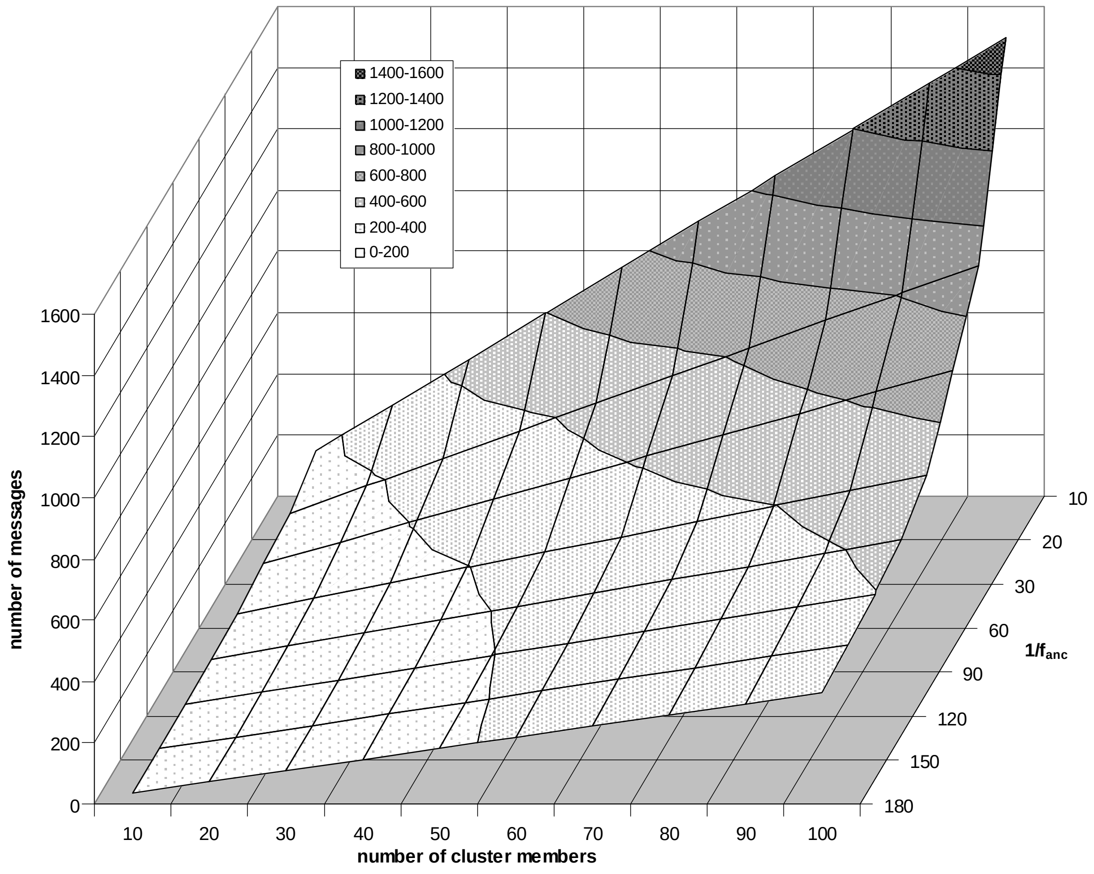


Figure 8.1: Message overhead for cluster formation

8.3.2 Energy Savings

The next performance metric we are interested in are the energy savings. The achieved energy savings depend on most of the system parameters. As an example, the mobility model has a large influence on the energy savings. Randomly moving nodes may not be clusterable and thus may not be able to save any energy at all. Nodes moving together as a group or very slowly are better suited for our system and thus produce better results. In general, we expect the mobility model parameters to have the highest impact on our system performance. It is impossible to give a complete analytical model for the node movements. Therefore, a more detailed discussion of the mobility model parameters must be done experimentally (see Section 8.4).

Deactivation Time

To model the energy savings, we first model the deactivation time, i.e., the time span during which a node can be deactivated and thus can save energy. For a cluster with n nodes, there are

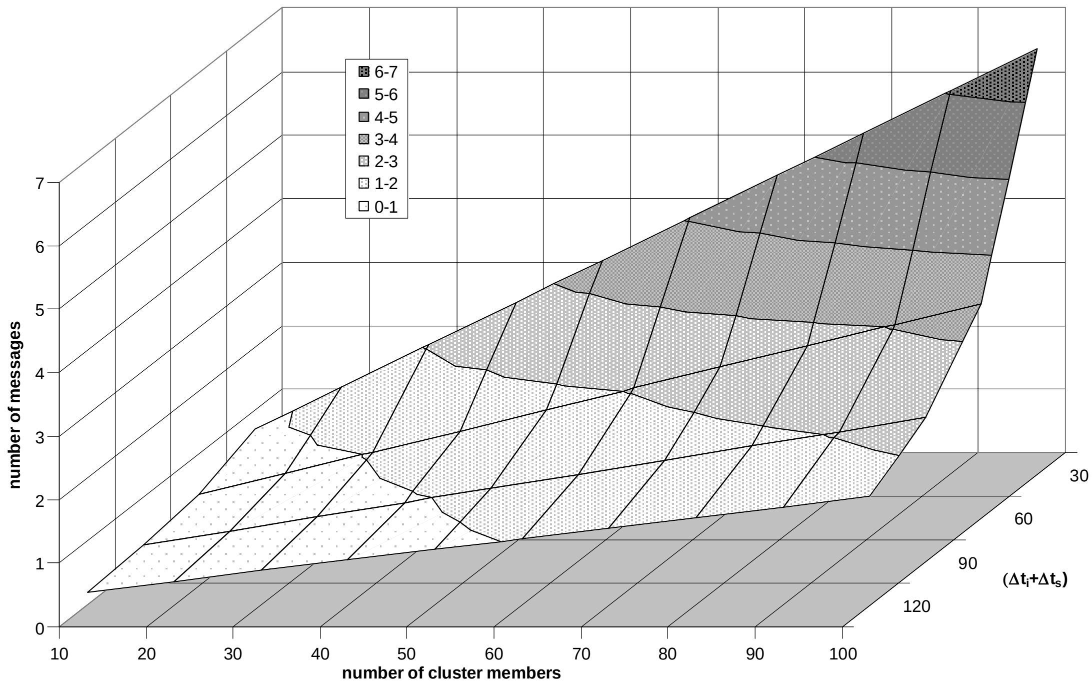


Figure 8.2: Message overhead (per second) for running cluster

always $(n-1)$ CNs and 1 CH, regardless of who the CH is. Therefore, we can ignore the CH rotation when computing the deactivation time and assume a fixed CH.

CNs can only be deactivated, if they are not in their probing phase. Therefore, we have to determine, how much time a CN spends probing and compute the remaining time. More specifically, we are interested in the fraction of time, that a CN spends not being in its probing phase. We can model this fraction $frac_{nProb}$ as:

$$frac_{nProb} := 1 - f_{prob} * \Delta t_{prob} \tag{8.10}$$

If a CN is not probing, it may have to stay activated because it is currently waiting for client requests. The remaining time is given as:

$$frac_{deact} := \frac{1}{\Delta t_i + \Delta t_s} * \Delta t_s \tag{8.11}$$

Lastly, a CN may not be deactivated if it is currently executing requests for clients. The amount of time a CN is not working is given by the idleness factor $frac_{idle}$ given in Table 8.3.

Using Equation 8.10, Equation 8.11 and $frac_{idle}$, the achievable deactivation time for a CN can be specified as:

$$frac_{save} := min(frac_{deact}, frac_{nProb}, frac_{idle}) \tag{8.12}$$

Energy Overhead

Of course, the deactivation time is not sufficient to compute the energy savings as we have omitted additional energy costs caused by our system so far.

Our first goal is to model the energy overhead per second of SANDMAN's continuous operation, which is caused by clustering messages. To do so, we reuse Equation 8.9 which specified the message overhead for this. We extend this equation with the sizes of the sent messages to get the overhead in byte. For better readability, we divide the resulting equation into multiple parts, each modelling a partial overhead.

The first partial overhead per second $BOPS_{anc}$ is caused by the CH sending announcements and is given as:

$$BOPS_{anc} := sizeof(ANC) * f_{anc} \tag{8.13}$$

The second partial overhead per second $BOPS_{sleep}$ gives the overhead for sending sleep messages and is modelled as:

$$BOPS_{sleep} := \frac{(n-1) * (sizeof(SLEEP\_ANC) + sizeof(SLEEP\_ACK))}{\Delta t_s + \Delta t_i} \tag{8.14}$$

The third partial overhead per second is $BOPS_{HO}$, which specifies the overhead caused by handing over the cluster regularly.

$$BOPS_{HO} := \frac{sizeof(HO\_REQ) + D_{avg} * S_{avg} * (n-1) + sizeof(HO\_ACK)}{\Delta t_{ch}} \tag{8.15}$$

Using these partial overheads, the total overhead per second in byte is given as:

$$BOPS_{cont} := BOPS_{anc} + BOPS_{sleep} + BOPS_{HO} \tag{8.16}$$

With Equation 8.16 we can now specify the energy overhead per second to operate a fixed cluster of size n as:

$$PO_{cont} := ((P_{send} - P_{idle}) + (P_{recv} - P_{idle})) * \frac{BOPS_{cont}}{d_{tx}} \tag{8.17}$$

Note, that each message is send and received by a member of the cluster. Therefore, we have to include energy consumption for both sending and receiving messages.

After modeling the continuous energy overhead of a fixed cluster, we extend our model with dynamic cluster changes. We include two kinds of changes: node joins and CH losses. We ignore lost CNs, as from the viewpoint of the cluster, this event normally causes no additional energy consumption. On the other hand, loosing a CH results in the cluster being dissolved and triggers a complete reclustering, consuming a lot of energy. To model this, we need two additional parameters. The first parameter a denotes the average number of cluster joins per second. It is used in Equation 8.24 on page 166. The second parameter b gives the average number of CH losses per second and is used in Equations 8.20 and 8.24.

To compute the energy cost per node join, we reuse Equation 8.6 for the message overhead of a join and modify it to give the overhead in byte for joining a new node as:

$$BO_{join} := sizeof(JOIN\_REQ) + D_{avg} * S_{avg} + sizeof(JOIN\_ACK) \tag{8.18}$$

Using this formula, we can model the total energy overhead for joining a new node as:

$$EO_{join} := ((P_{send} - P_{idle}) + (P_{recv} - P_{idle})) * \frac{BO_{join}}{d_{tx}} \tag{8.19}$$

When a CH is lost, its cluster is disbanded and all CNs switch back to UN mode. After that, a completely new clustering is started. Therefore, for each CH loss, $(n-1)$ nodes must join a new cluster, resulting in an additional energy overhead of $(n-1) * EO_{join}$. In addition, all nodes are awake and can thus not save energy. To take this into account, we model the additional awake times as energy costs to substract them from the energy savings. However, we cannot simply assume that the nodes would have been deactivated without the lost CH, as they could be active due to other reasons, e.g., a low $frac_{idle}$ value. Therefore, we reuse Equation 8.12 to determine the upper bound of the deactivation time that we can substract:

$$frac_{loss} := min(b * \Delta t_{avg}, frac_{save}) \tag{8.20}$$

The additionally consumed energy per second for the whole cluster is thus:

$$PO_{loss} := (n-1) * frac_{loss} * (P_{idle} - P_{sleep}) \tag{8.21}$$

Finally, while being awake, the nodes send announcement messages, resulting in an additional message overhead in byte of:

$$BO_{anc} := n * \Delta t_{avg} * f_{anc} * sizeof(ANC) \tag{8.22}$$

As announcement messages are broadcast and therefore received by all nodes in the neighborhood, the energy overhead of sending n announcements is:

$$EO_{anc} := \frac{BO_{anc}}{d_{tx}} * (P_{send} - P_{idle}) + \frac{BO_{anc}}{d_{tx}} * (P_{recv} - P_{idle}) * n \tag{8.23}$$

Now, we can specify our formula for the total energy overhead per second of a running cluster with dynamic cluster changes as:

$$PO_{total} := PO_{cont} + a * EO_{join} + b * ((n-1) * EO_{join} + EO_{anc}) + PO_{loss} \tag{8.24}$$

Energy Savings

After modeling both the deactivation time and the energy overhead, we now can specify a formula for the actual energy savings. To stay independent of the system execution time, we give the energy savings per second. Using Equations 8.12 and 8.24, we can model the energy savings per second for a cluster with n nodes as:

$$P_{save} := (n-1) * frac_{save} * (P_{idle} - P_{sleep}) - PO_{total} \tag{8.25}$$

This formula allows us to investigate the achievable energy savings depending on a number of parameters. First, we show the energy savings per second and node for different cluster sizes n and deactivation lengths Δt_s in a stable system without fluctuations, i.e., with $a = b = 0$ (see Figure 8.3). We used relative values for the energy savings to be independent of the actual energy consumption. The different parameters were set according to Table 8.5 on page 175. The achieved savings are between 75% and 90%. Note, that our formula assumes an already formed cluster, i.e., we do not consider the initial overhead for cluster formation. We chose to omit the initial overhead, as for long running systems it is neglectable when compared to the continuous savings. The continuous overhead for cluster maintenance is included in our formula. Concerning the results, we can see that although different cluster sizes clearly have an effect and larger clusters are able to save more energy, the differences are lower than one would expect, e.g., between about 75% and 83% for $\Delta t_s = 10s$. This is the case as the smallest cluster size used here is 10 nodes, which already results in the nodes being deactivated for nearly 90% on average, compared to about 98% for clusters with 100 nodes. With increasing deactivation lengths Δt_s, the energy savings rise up, as expected.

As we can see, it is possible to get very high energy savings with SANDMAN. However, the savings discussed so far were achieved in stable environments without any clustering changes.

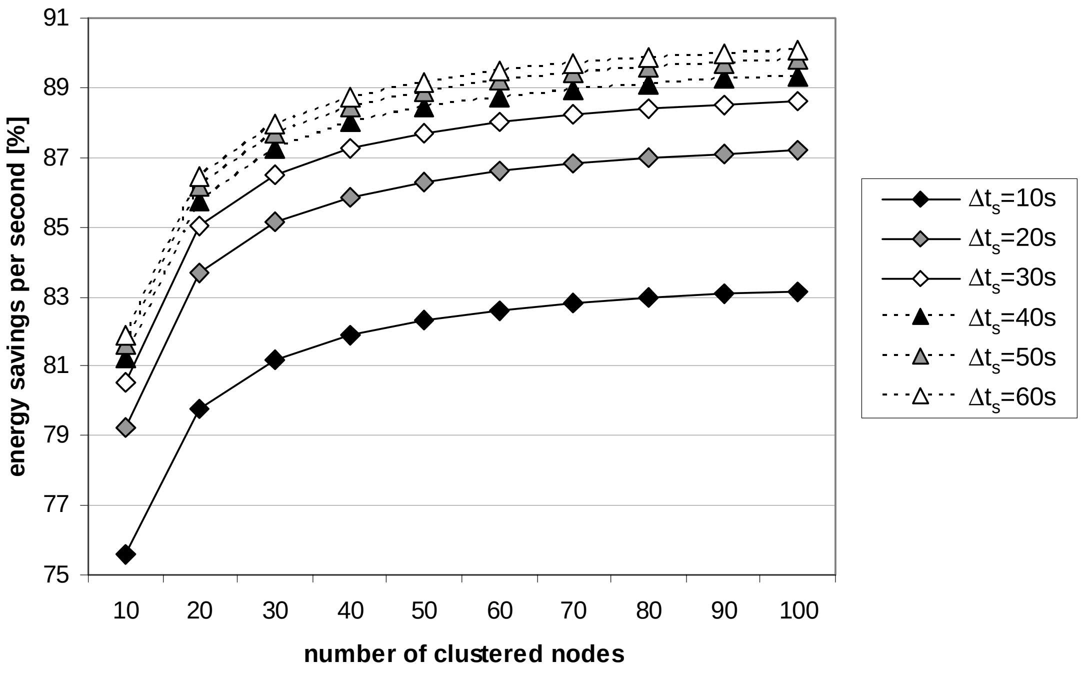


Figure 8.3: Energy savings for different cluster sizes and deactivation lengths

To get a better understanding of the influence of fluctuations on the savings, we additionally computed the energy savings for increasingly unstable clusters. We chose to vary the mean time until a CH is lost – our parameter b – between $0\frac{1}{s}$ and $\frac{1}{30s}$ for different cluster sizes. The results can be seen in Figure 8.4. Clearly, different stability levels have a large impact on the achieved savings. While savings of about 90% are normal for the stable system, the benefit of SANDMAN in terms of energy efficiency decreases to about 64-58% for an average CH loss every 5 minutes and 39-36% if the CH is lost every 2.5 minutes on average. If the system is even more instable, SANDMAN is not able to save any energy. Instead, it consumes more energy than an idle system, resulting in negative savings of up to -2%. Clearly, this shows that SANDMAN should not be used in such unstable environments.

8.3.3 Discovery Precision and Recall

Defining an analytical model for discovery precision and recall is a difficult task, as they are highly dependent on the current node mobility. Therefore, we can only create a very simplified analytical model.

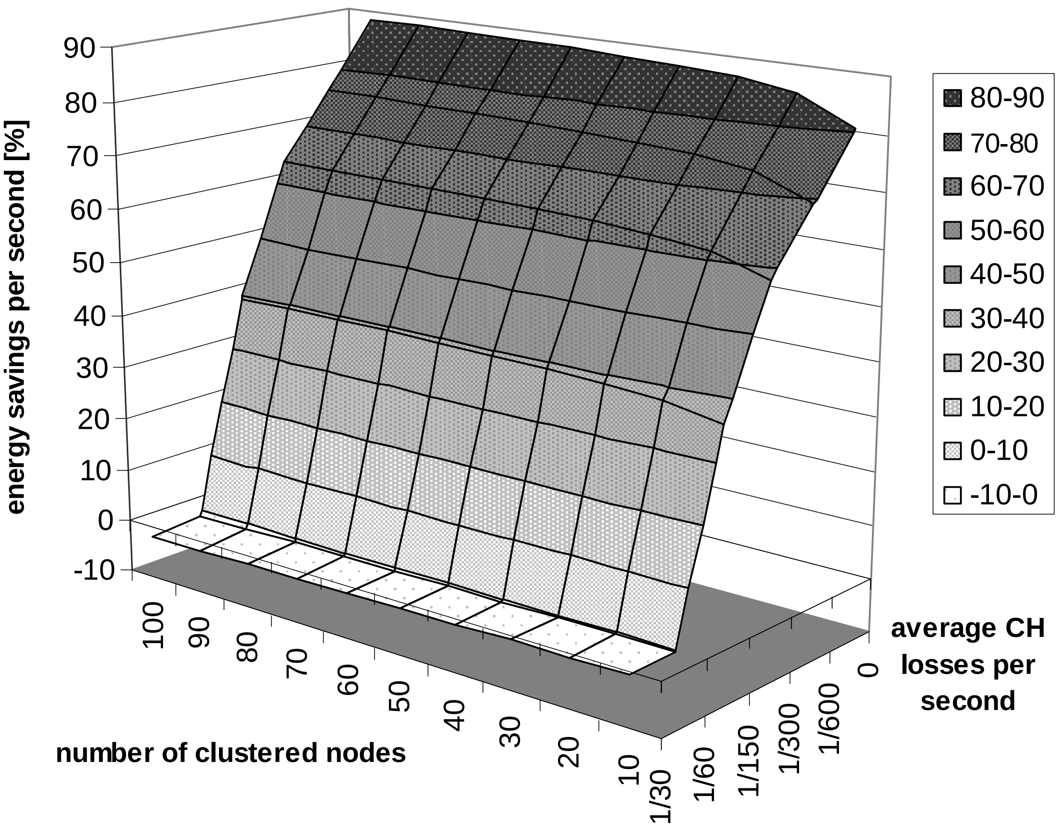


Figure 8.4: Energy savings for different cluster sizes and stability levels

Discovery Precision

To model the discovery precision, we introduce a new parameter c which denotes the average number of CNs per second that leave the communication range of their CH. In case of such a CN loss, the average time until the loss is detected is $\frac{lease_{avg}}{2}$. Until then, the CH will still announce the CN and its services to clients. In the worst case, clients are interested in all services of the cluster, leading to the CH announcing $c * \frac{lease_{avg}}{2} * S_{avg}$ wrong services in average. In other words, for a cluster of size n, the CH should announce $(n-c) * \frac{lease_{avg}}{2} * S_{avg}$ services, as long as there are nodes left in the cluster. However, it falsely announces $n * \frac{lease_{avg}}{2} * S_{avg}$ services. This results in a discovery precision of:

$$p := \frac{n * S_{avg} - c * \frac{lease_{avg}}{2} * S_{avg}}{n * S_{avg}} \tag{8.26}$$

Using this formula, we can compute the discovery precision for different cluster sizes n and different stability factors c, as shown in Figure 8.5. We used cluster sizes of one, four and ten nodes, similar to the experimental settings used later. The stability was varied between totally

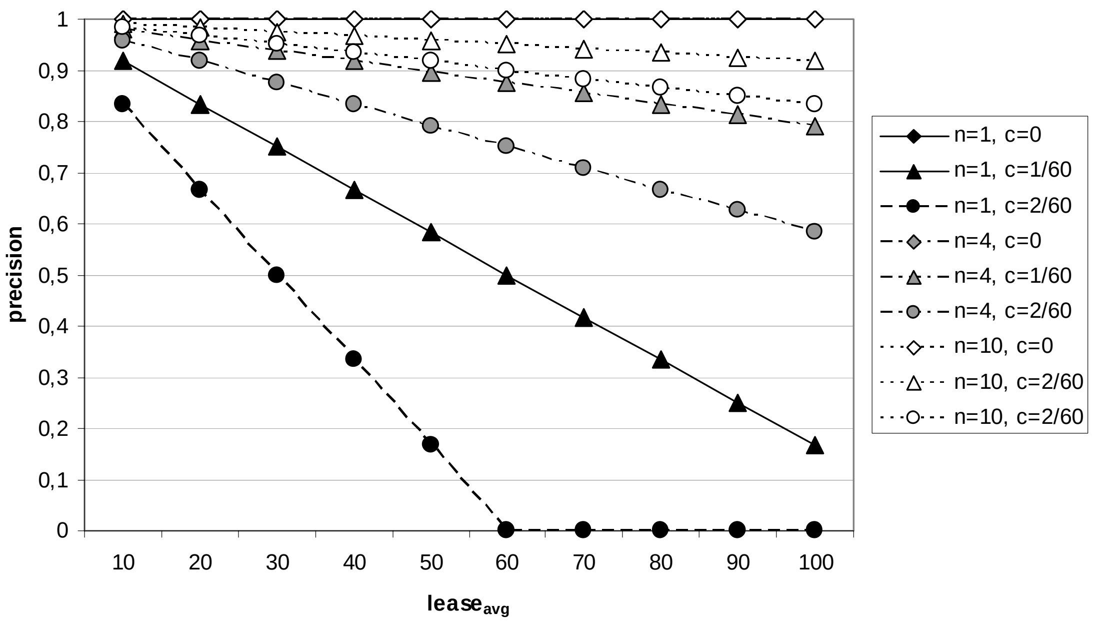


Figure 8.5: Precision for different cluster sizes and stabilities

static ($c = 0$) and one, respectively two CN losses per minute. These are rather high values, as in most cases the cluster management will produce much more stable clusters. However, the precision for such stable clusters would vary little from the static situation. The presented results show that the precision decreases linearly with the average lease length $lease_{avg}$. For static situations, the precision is always $p = 1.0$, as expected. With higher fluctuation, the precision drops significantly. A special case can be seen for the setting with two node losses per minute and a cluster size $n = 1$[1]. Starting with a lease length of $30S$, there may not be any services left that should be announced, i.e., all announced services are wrong, leading to a precision $p = 0$. The cluster size influences the results significantly. This is the case, because a lost CN is less important in large clusters than in small ones.

Discovery Recall

The discovery recall is affected if a CN leaves its CH's Smart Peer Group without realizing it, e.g., because it is currently deactivated. Until the CN detects the CH loss, it will not announce itself or its services to clients. Thus, its services are undiscoverable for clients. For our analytical model, we define the parameter d to give the average number of CNs per second, which become newly available for communication with a node, and whose CH cannot communicate with this node. An example for this is a node that switches between two Smart Peer

[1]Note, that we expect the cluster size to be set back to n regularly after the loss was detected, e.g., due to new nodes joining, to normalize the results for a given cluster size.

Groups. The services of these CNs cannot be discovered in the new environment, as long as the CNs do not detect the changed situation and switch back to UN mode. Assuming a normal distribution, this situation lasts for an average time of $\frac{lease_{avg}}{2}$, leading to an average number of undiscoverable services of $d * \frac{lease_{avg}}{2} * S_{avg}$. That means that a given client discovers only the nodes available in the current cluster, i.e., $n * S_{avg}$ for a cluster size of n nodes, instead of $n * S_{avg} + d * \frac{lease_{avg}}{2} * S_{avg}$. This leads to the following formula for the discovery recall:

$$r := \frac{n * S_{avg}}{d * \frac{lease_{avg}}{2} * S_{avg} + n * S_{avg}} \tag{8.27}$$

Similar to the discovery precision, we show the discovery recall for different cluster sizes n and different stability factors c in Figure 8.6. Again, we used cluster sizes of one, four and ten nodes, and varied d between zero, one and two new nodes per minute. Note, that the

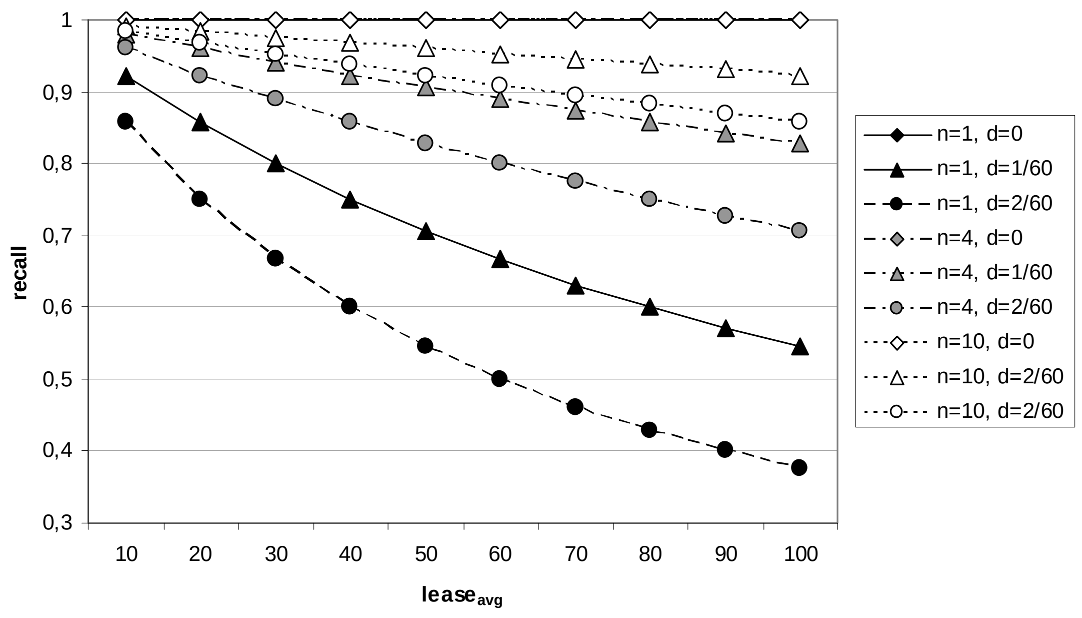


Figure 8.6: Recall for different cluster sizes and stabilities

recall seems to be more resilient to fluctuations than the precision. It never falls to zero in the presented settings. However, this is due to the different number of currently available nodes in both cases. For the precision, we assumed to loose nodes, effectively lowering the cluster size temporarily below n. In case of the recall, we expect additional nodes to show up, i.e., we have more than n nodes offering services. Other than that, the results for the discovery recall follow the same patterns than the precision's ones. For no fluctuations, the recall is always $r = 1.0$. The higher the fluctuation, the lower the recall. The same is true for the lease length and the cluster size, as before.

8.3.4 Discovery Latency

SANDMAN makes sure that despite deactivated CNs, discovery requests can always be answered by CHs. CHs themselves are never deactivated. Therefore, the resulting discovery latency is independent of the deactivation time Δt_s and identical to that of a pure peer-based discovery system without node deactivations. The only factors contributing to it are the communication latency per hop and the network diameter in hops. This is one of the main advantages of SANDMAN compared to an approach which clusters on the network layer. The latter must buffer discovery requests at the CHs and wait for the CNs to awake to answer the requests. In SANDMAN, the CHs can anwer the requests immediately.

8.3.5 System Parameterization

As we have discussed before, the analytical model can be used to parameterize SANDMAN in such a way that certain properties are achieved, depending on the given scenario. To demonstrate this, we present in this section a scenario and derive a suitable parameterization for it.

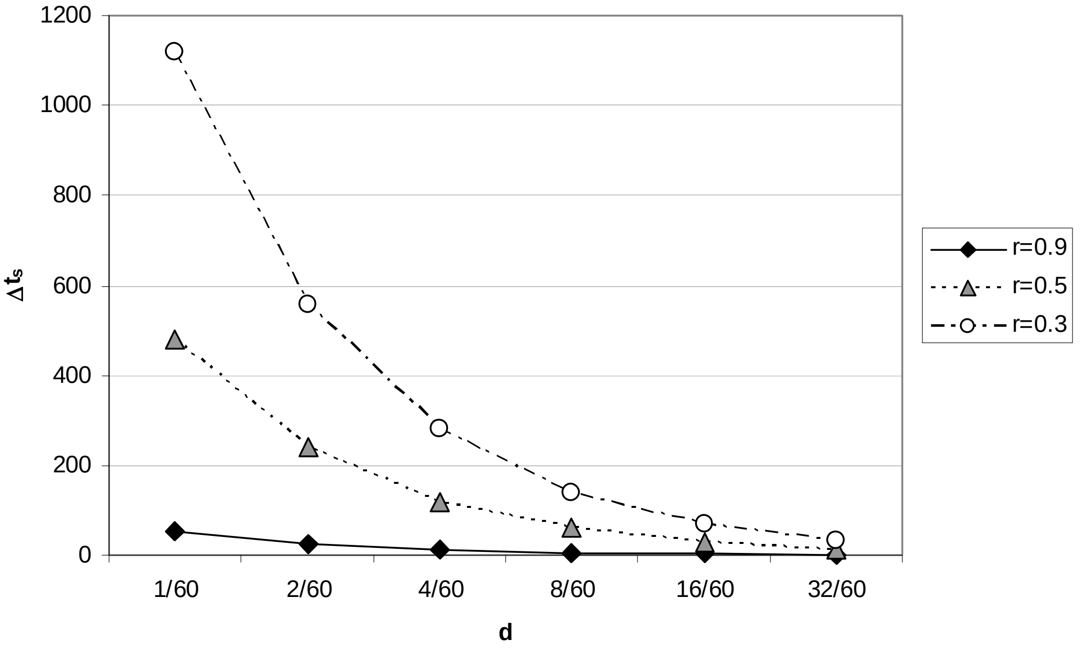


Figure 8.7: Recall for different cluster sizes and stabilities

In our scenario, we want to set the deactivation times according to a given discovery recall. This could, e.g., be useful, if we want to adjust our system to environments with different numbers of services. If we are in a resource-rich environment that contains a lot of services, it would be acceptable to experience a certain loss in discovery recall. On the other hand, in a resource-poor environment with few services, we want to maximize the recall to discover as much services as possible. Given Equation 8.27 and Equation 8.4, we can compute the achievable deactivation times as $\Delta t_s = \frac{2}{d} * (\frac{n}{r} - n) - (\Delta t_i + lease_{tol})$. The result of this is shown in Figure 8.7 for a cluster size of $n = 4$ and different stability levels d. In the resource-poor environment, we want to achieve a recall of at least 0.9. This leads to a deactivation time Δt_s of about 52 s for $d = \frac{1}{60}$ and less than a second for $d = \frac{32}{60}$. However, if the environment contains more services, a lower recall is acceptable. For $r = 0.5$, we can deactivate the devices for 479 s in the more stable environment. If a recall of 0.3 is acceptable, Δt_s can be as high as 1119 s.

8.4 Experimental Evaluation

In the following, we describe the experiments performed to evaluate the SANDMAN system and discuss their results. To do so, we first explain the overall setting used for our experiments. Then we present a number of different scenarios and discuss the experiment's results, focusing on the different performance metrics described in Section 8.1.

8.4.1 Emulation Environment

To perform our experiments, we use the Network Emulation Toolkit (NET) [HMTR04]. NET is a Linux-based emulation environment for testing and evaluating network protocols. A set of emulation tools, e.g., the so-called net shaper, allows the user to set up different evaluation environments, like a local area network, a satellite-based communication system, or a MANET. NET is based on a cluster of 64 Linux-PCs, which are interconnected via a GB network. In contrast to simulation environments, an emulation environment like NET allows to test unmodified real-world implementations, resulting in more realistic results. In addition, measurements using real devices are highly restricted in size due to the number of available mobile devices. NET allows to define virtual mobile devices, which can be used for evaluating large scenarios. Therefore, NET is an ideal environment to evaluate SANDMAN.

However, some enhancements to the existing NET system were needed because NET provided only rudimentary support for energy-efficient systems. Although NET supports wireless communication technologies, like IEEE 802.11, there is no support for different energy modes of a node or communication interface. NET offers a simple battery model (see [Sto04]) but does not model the energy consumption of different components, specifically the network interface.

Node Deactivation

NET offers no support for deactivating emulated devices or network interfaces. However, the Linux kernel offers functions to deactivate and activate a network interface programmatically.

In order to evaluate our approach, a software tool was developed that allows to access these functions from within BASE by executing this tool and passing all needed information as command line parameters. This solution allows to physically deactivate a network interface and therefore guarantees a realistic system behavior, as no communication on any lower layer can be performed while deactivated.

Energy Consumption Modeling

The possibility to measure the actual energy consumption of our system is crucial for the evaluation. The continuous energy consumption of a node in its different energy modes is as important as the energy consumed to send and receive a certain amount of data. In addition, to allow SANDMAN to request the current battery level of a node at runtime, energy consumption measurement must be performed at runtime. There are two basic approaches to realize such a model. First, we can extend NET with full support for measurement and integrate this with the NET battery emulation. Second, we can add measurement support in BASE and extend NET only as much as needed. We chose the second approach because it allows us more control over the measurement and requires less changes to the NET emulation environment. We measure continuous consumption by integrating a special battery component into BASE that keeps track of the current system life time and energy mode and the durations in these different modes. Using this data, the component can compute the continuous consumption at any time. Communication related consumption is measured by reading the amount of send and received bytes and packets which is provided by Linux in the special file `/proc/net/dev` in the *proc* file system. This allows us to measure the actual consumption even if a lower layer has send the data, resulting in more realistic results. However, only unicast communication is included in this file. Therefore, NET was extended to offer an additional file (`/proc/net/netshaper_energy`) to provide the same information for broadcast communication. Using this information, the BASE battery component computes the energy consumption for sending and receiving data, integrate it with the continuous consumption so far and report the total consumption at any time during runtime.

Mobility Model

As already discussed, we use the RPGM model. To create concrete RPGM movement traces, we use BonnMotion [Uni05], a mobility trace generator developed at the Universität Bonn. BonnMotion is able to create mobility traces using different models, e.g., random waypoint or RPGM. NET contains a basic scripted simulation control environment. We included BonnMotion into this environment in order to generate mobility traces conveniently.

8.4.2 Experimental Settings

After we discussed in Section 8.3 how different system parameters influence our performance metrics, we present a number of scenarios used in our experiments. These scenarios offer different node mobility patterns. As already discussed, it is not sufficient to model mobility only with speed. Instead, we vary two parameters: movement speed and the number of nodes per group.

For our experiments we choose three characteristic movement speed values v_{grp}: stationary, pedestrian, and car traffic. Clearly, in stationary scenarios nodes have a movement speed of 0 m/s. For pedestrian movement we use 2 m/s and for car traffic 15 m/s. Perpendicular to the movement speed is the number of nodes per group. We choose three characteristic group sizes n_{grp}: a single person moving randomly through town, a family moving together, and a tourist group. Each person carries one device and is therefore modeled as one node. Clearly, a group size of one leads to the well known random waypoint model. Families are modeled as groups of 4 nodes, tourist groups contain 10 nodes. Note, that in our settings a group of, e.g., 4 persons moving together is equivalent to a single person carrying 4 devices. Therefore, we omitted further scenarios with multiple devices per person.

The combination of these three movement speeds with the three group sizes leads to 9 scenarios, named Scenario A to I, which are presented in Figure 8.8. For each scenario we created ten movement traces that were used in our measurements.

	Random Waypoint	*Family (4 person groups)*	*Tourist group (10 person groups)*
Stationary	Scenario A	Scenario B	Scenario C
Pedestrians	Scenario D	Scenario E	Scenario F
Car Traffic	Scenario G	Scenario H	Scenario I

Figure 8.8: Experimental evaluation scenario overview

In addition to the mobility model parameters, we vary the deactivation time Δt_s between 30 and 150 seconds, in steps of 30 seconds. Note that these values only affect the delay for the first usage of a service by a client, since SANDMAN keeps nodes offering services that are used by at least one client awake. While an initial maximum usage delay of 30 s is acceptable for accessing devices like, e.g., a printer, 150 s is a reasonable value for, e.g., home automation devices like a heating control actuator. Further system parameters used in our experiments are listed in Table 8.5.

Parameter	**Setting**
w	300 m
h	300 m
n	40
r_{tx}	100 m
d_{tx}	10 Mb/s
Δt_{grp}	0 s
σ_{grp}	0
$dist_{grp}$	0 m
p_{grp}	0
Δt_{act}	0 s
Δt_{deact}	0 s
P_{send}	1400 mW
P_{recv}	950 mW
P_{idle}	805 mW
P_{sleep}	60 mW
E_{deact}	0 mW
E_{act}	0 mW
S_{avg}	1
$frac_{idle}$	1.0
γ	60 s
Δt_i	1 s
$lease_{tol}$	2 s
$lease_{avg}$	$\Delta t_s + \Delta t_i + lease_{tol}$
$lease_{max}$	$lease_{avg}$
f_{anc}	1/20
Δt_{join}	60 s

Table 8.5: Parameter settings for experimental evaluation

The energy values are identical to the ones already used in Table 3.1. The lease values are computed according to Equation 8.4 on page 159.

8.4.3 Experimental Results

We performed measurements in all scenarios given before. Each measurement was done with ten different precalculated mobility traces and the results were averaged. In the following we present the most characteristic results for each of our performance metrics.

Message Overhead

Our first diagram shows the total message overhead per second (see Figure 8.9). We present the results of our measurements for Scenarios D, E, and F. That means, we compare the message overhead for different group mobilities with pedestrian movement speed and relate it to the sleeping intervals.

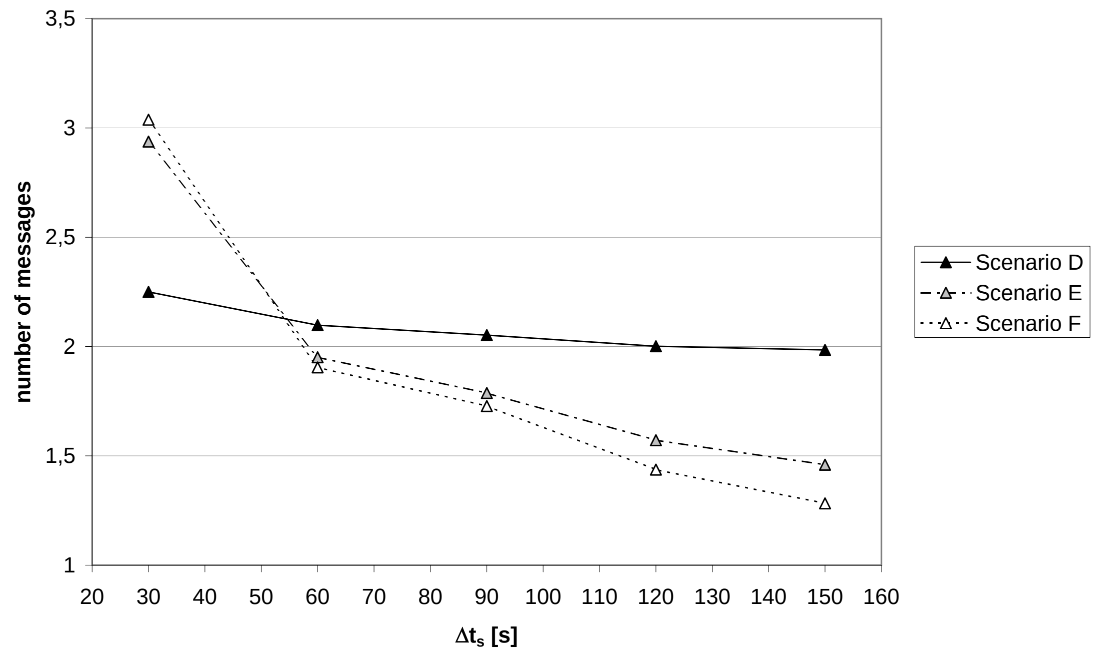


Figure 8.9: Total message overhead per second

All three scenarios show the same tendency, the message overhead decreases with increasing Δt_s. This is exactly what we expected, since with higher values for Δt_s the CNs must send sleep announcements less often. This effect becomes increasingly apparent for larger group sizes, as more nodes are clustered.

Interestingly, for $\Delta t_s = 30s$ the message overhead is higher for larger group sizes. The reason for this is that with this Δt_s CHs and UNs have a lower message overhead than clustered CNs due to our announcement frequency f_{anc} being set to $1/20$. A node announcing itself sends one ANC message each 20 seconds. A clustered node sends one SLEEP_ANC each $\Delta t_s = 30s$ and receives a SLEEP_ACK message, effectively needing an average of one message each 15 seconds. Therefore, in this special case, the scenarios with less CHs and more UNs are cheaper with respect to their message overhead. Further measurements show that the message overhead for the other scenarios follows the same tendency.

Energy Savings

Concerning the achievable energy savings, our results show that – as expected – savings depend on Δt_s, group size, and movement speed. Higher Δt_s lead to longer sleep times and – as shown before – lower message overhead. Similarly, larger group sizes result in larger clusters and thus the CH's overhead for cluster management is distributed better. This can be seen, e.g., in Figure 8.10. Here, we show the average energy savings per node for scenarios with pedestrian movement speed and random waypoint mobility (Scenario D), groups of size 4 (Scenario E), and groups of size 10 (Scenario F) depending on different Δt_s.

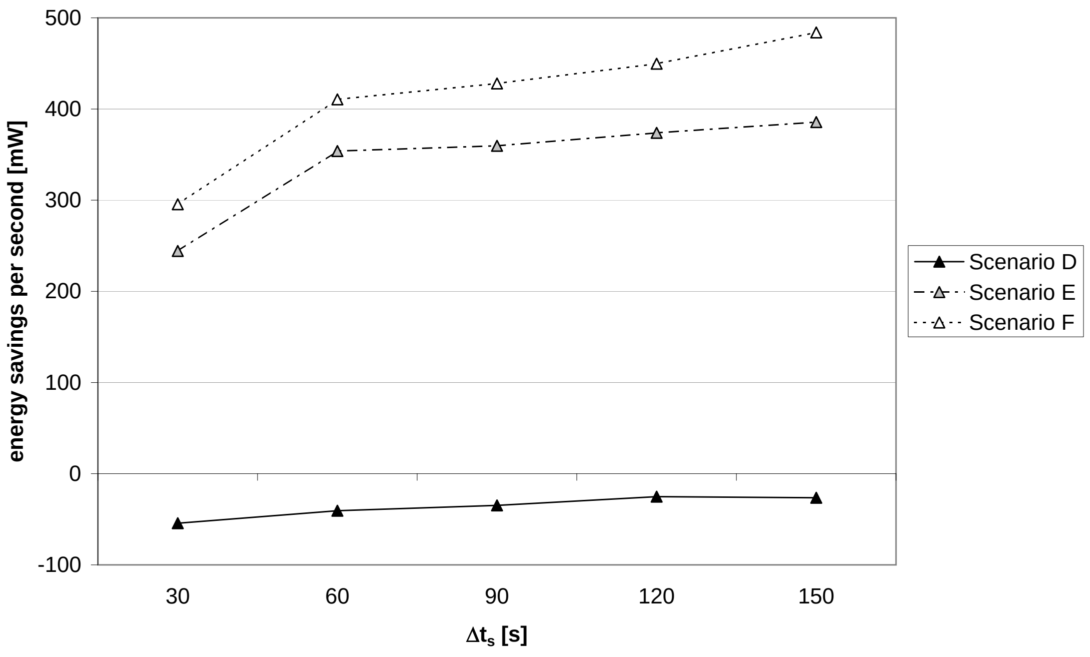


Figure 8.10: Energy savings

Similar to the results for the message overhead, the energy savings increase faster between $\Delta t_s = 30s$ and $60s$ than for larger values of Δt_s. This is due to the message overhead in this interval. For larger Δt_s, the savings increase slower but linear, as expected.

In Scenario D, the nodes consume more energy than without SANDMAN. This is due to the fact that nodes are clustered rarely and the message overhead consumes more energy than is saved by deactivating nodes. Therefore, for this scenario, SANDMAN is not beneficial and should not be used. However, for larger group sizes, the nodes are able to save up to 484 mW per node for $\Delta t_s = 150s$ and a group size of 10. For the chosen continuous node consumption of $\Delta t_{idle} = 805mW$, this is a saving of approximately 60% per node, including CHs and UNs. For scenarios with other movement speeds, the results are accordingly, while total values for higher speeds are lower. This is the case, as with higher mobility clusters become less stable and nodes must recluster more often. We can observe the same effect when comparing scenarios with identical group sizes but different movement speeds (e.g., Scenario A, D, and G). In all three such cases, the achieved energy savings are lower for higher speeds.

Discovery Precision

The discovery precision shows how exact the system answers a discovery request. If services that are not available anymore are reported back to the requesting client, the discovery precision decreases. Clearly, we expect the discovery precision to decrease with increasing deactivation periods Δt_s and increasing mobility, both in terms of movement speed and group sizes. Note, that if no nodes are deactivated or if nodes do not move, the discovery precision should be one.

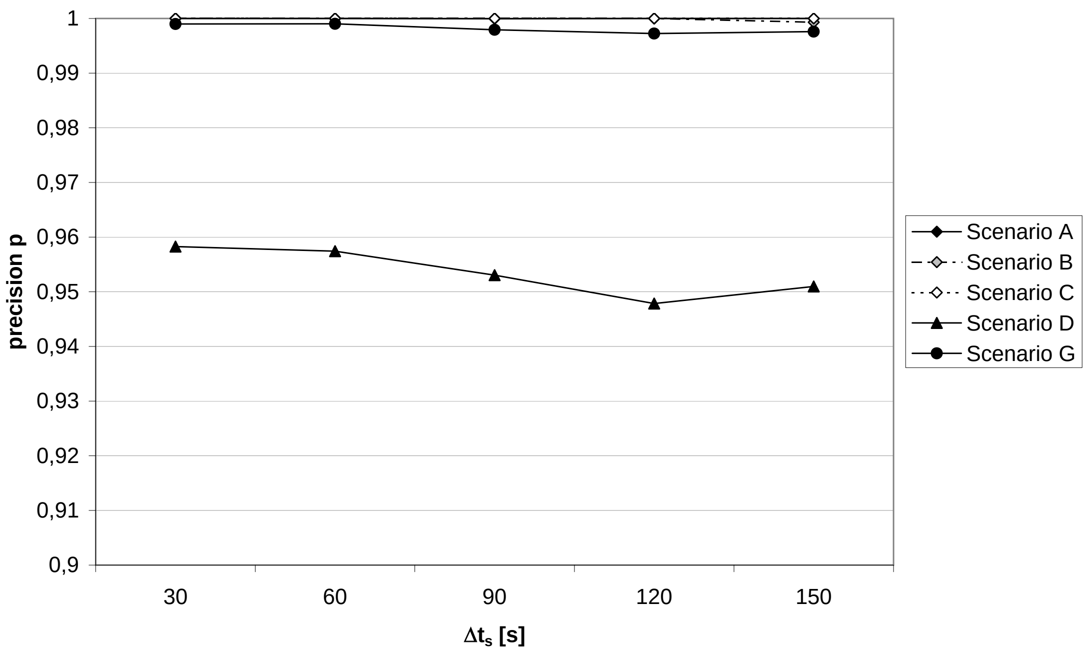


Figure 8.11: Discovery precision

As can be seen in Figure 8.11, the average precision for stationary scenarios (Scenarios A,B,C) varies between 1.0 and 0.999302832. This is the case because in these scenarios clusters are highly stable and nodes do not get lost due to mobility. On the other hand, in highly mobile scenarios like Scenario G nodes move very fast and are clustered rarely. Hence, SANDMAN effectively behaves like a normal peer-based discovery system and the discovery precision is near 1.0 again. In Scenario D, nodes move slow enough to get clustered more often. However, as the nodes move separately, these clusters must be dissolved frequently. Therefore, the discovery precision drops to lower values, e.g., 0.947854464 for $\Delta t_s = 120s$.

Discovery Recall

The discovery recall is negatively influenced by nodes loosing their CHs without noticing it. In Figure 8.12 we show the discovery recall for the random waypoint scenarios A, D, and G.

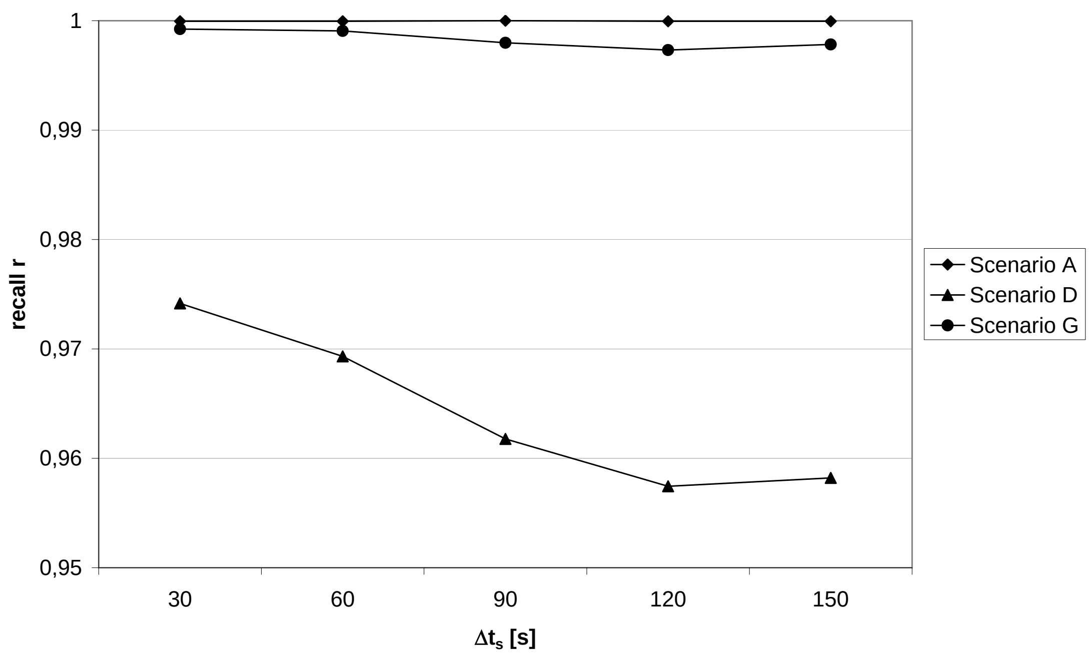


Figure 8.12: Discovery recall

Similar to our measurements of the discovery precision, the discovery recall stays near 1.0 for the stationary Scenario A, where nodes are not lost, and the highly mobile Scenario G, where nodes are rarely clustered. Again, for pedestrian movement speed, the recall drops to lower values as clusters are formed and dissolved more frequently.

Discovery Latency

As already mentioned in Section 8.3.4, the discovery latency depends on the communication delay and is independent of the duration of the deactivation time Δt_s. Therefore, we can choose different values for Δt_s without affecting the discovery latency. In addition, the latency is mostly independent of the current clustering of the nodes, as requests are never delayed due to nodes being clustered. Consequently, the discovery latency is the same for all our different scenarios.

This distinguishes SANDMAN from systems that make clustering and deactivation decisions on the network layer. An example for this would be a pure peer-based service discovery system in combination with an energy efficient routing protocol. In such a system, messages that are sent to currently deactivated nodes are buffered at the CHs until the nodes are reactivated. Thus, service discovery messages are delayed until their receivers awake and the desired energy savings must be balanced with the discovery latency. In SANDMAN, discovery requests are answered immediately by the CHs, decoupling energy savings and discovery latency and therefore enabling long deactivation periods in combination with prompt service discovery.

To further analyze this advantage of SANDMAN, we measured, how much faster a discovery is performed in SANDMAN compared to a system that buffers discovery messages. The results are shown in Figure 8.13 for different group sizes and in Figure 8.14 for different movement

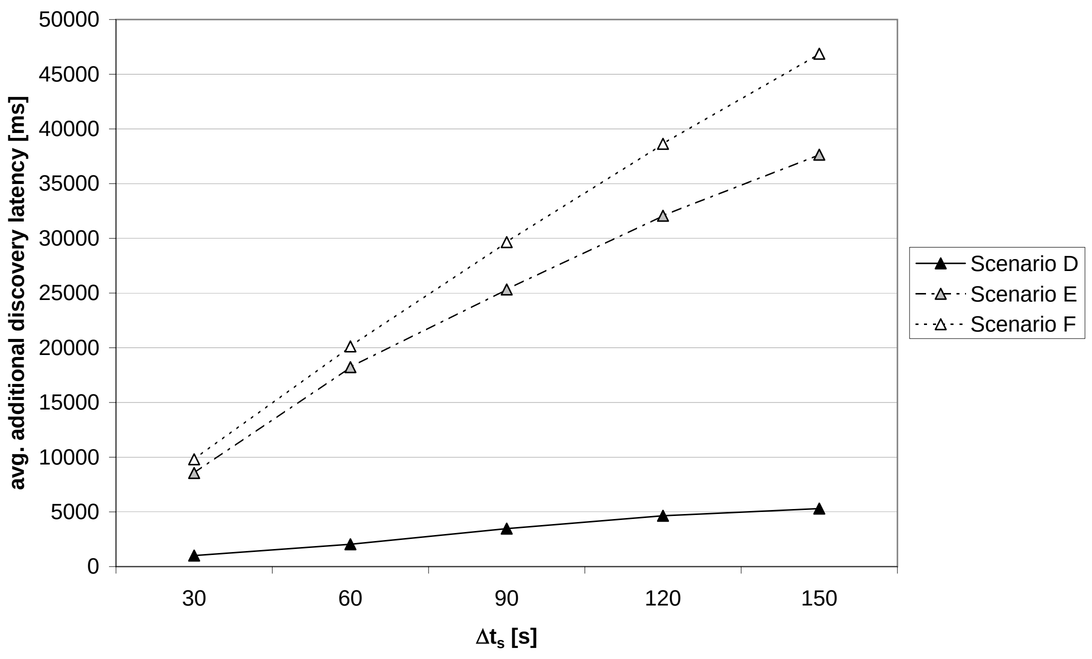


Figure 8.13: Discovery latency overhead for not using SANDMAN (1)

speeds, both for varying Δt_s. Note that the diagrams show the additional latency of not using

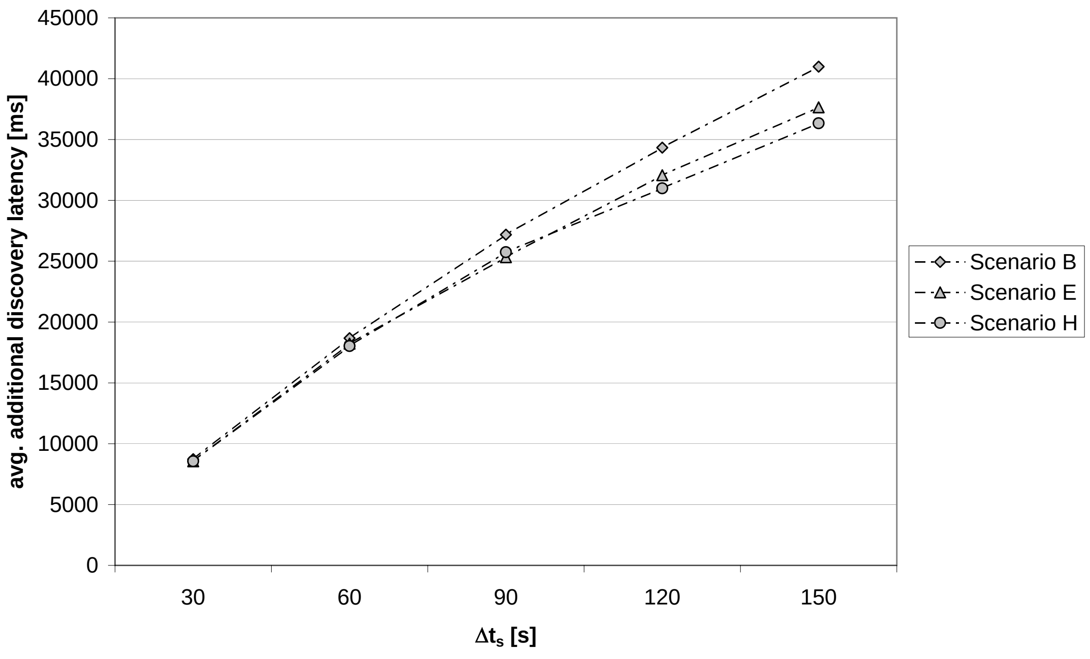


Figure 8.14: Discovery latency overhead for not using SANDMAN (2)

SANDMAN, instead of the absolute latencies of discovery requests. As expected, the discovery latency rises with Δt_s in all scenarios. However, it does not reach the theoretical value of $0.5 * \Delta t_s$, since active nodes, i.e., UNs and CHs, answer the discovery requests immediately and lower the average latency. This can be seen especially for Scenario D as in this scenario less nodes are clustered and deactivated than in the other scenarios. Therefore, the average discovery latency is much lower.

Note, that these results can also be interpreted as the average latency before a client can actually use a previously discovered service. After discovering a service, a client has to wait for the node providing the service to be reactivated. At this time, the client can contact the service. This latency is the same for SANDMAN and other approaches, which cluster on lower layers, as clients have to wait for the service provider's reactivation in both cases. Therefore, SANDMAN does not enable client to use services faster but only to discover them faster.

8.5 Summary and Conclusion

In this chapter, we have analyzed the influence of different system parameters on our service discovery system SANDMAN. To do so, we developed an analytical model to investigate the behavior of SANDMAN in different settings. However, one of the most important influence factors, the mobility model, can not be modeled completely analytically. Therefore, we additionally performed a number of experiments in which we varied the movement speed and group size to evaluate SANDMAN in different mobility settings.

SANDMAN is designed to save energy in environment with slowly moving nodes or nodes moving together in groups. For such environments our results show that SANDMAN is able to save substantial amounts of energy of up to 60%. With increasing node movement speed, energy savings get lower. However, SANDMAN was able to save energy for all scenarios involving group mobility. As expected, SANDMAN clusters nodes less often in random waypoint scenarios. This is due to our clustering algorithm which specifically tries to cluster nodes with low relative movement, only. Therefore, nodes cannot be deactivated in these scenarios and thus do not save energy. Due to the additional message overhead of SANDMAN, this leads to negative energy savings in the random waypoint scenarios, i.e., our system consumes more energy than a classical peer-based discovery system without node deactivations. In such cases, SANDMAN should not be used.

9 Conclusion and Outlook

9.1 Conclusion

In this thesis, we presented a novel peer-to-peer-based approach to establish Pervasive Computing systems as dynamic sets of spontaneously cooperating devices, called Smart Peer Groups. Smart Peer Groups allow cooperation between devices without external infrastructure support and can therefore be used in a multitude of scenarios. They are challenged by the heterogeneity of the participating devices and communication networks, the system dynamism concerning the availability of devices and resources, and the resource constraints of the cooperating devices namely computational power, memory and energy. To meet these challenges, Smart Peer Group-based systems must offer means to monitor their surroundings continuously, to detect changes and to react accordingly, e.g., by switching to another communication technology or reselecting the used services.

In this thesis we have analyzed the specific requirements of Smart Peer Group-based systems and developed concepts and algorithms to provide the flexible and efficient discovery and usage of remote functionality, modeled as services.

To access remote services, we have developed and presented our micro-kernel-based middleware BASE. BASE decouples the communication model of an application from the synchronization model of the used interoperability protocol and allows to dynamically select between different communication technologies and protocols as needed. In addition, BASE is able to adapt its own resource requirements to the resources available at its host, allowing its usage on a large variety of devices.

To dynamically discover the devices and their services that are available in a given scenario, we have developed our service discovery system SANDMAN. SANDMAN uses a clustering scheme to provide highly accurate service discoveries in combination with temporarily deactivated devices to save energy. Different algorithms for cluster management and deactivation scheduling can be combined with SANDMAN to adapt it to different system environments. In this dissertation we have presented examples for algorithms for cluster management and scheduling.

9.2 Outlook

While the concepts and algorithms developed for BASE and SANDMAN provide a solid base for the creation of Smart Peer Group-based systems, there are a number of further research directions that could be explored in the future.

First, additional algorithms for cluster management and deactivation scheduling could be developed. As an example, cluster management protocols for multi-hop or hierarchical clusters could allow to include more nodes in a single cluster. Additionally, deactivation scheduling algorithms could use more complex prediction algorithms to detect more complex usage patterns of services and deactivate the nodes accordingly.

As a second future research direction, the CHs can be used to provide other system services to their clusters. As an example, the CHs can be used to compute suitable application configurations for automated adaptation. As shown in [HHSB07] a centralized application configuration, in which a single device computes the whole configuration of an application may be beneficial in many scenarios to a decentralized peer-based approach. The CH can be chosen to act as the central configuration manager. Clearly, this scenario poses new requirements on the clustering and therefore demands new algorithms, e.g., electing CHs depending not only on their remaining lifetimes but their computational power and memory size. In addition, the CH can be used to decouple clients and service providers by masking the identity of individual services toward their clients. Instead, the clients interact solely with the CH without any knowledge about the nodes actually executing the services. This allows the CH, e.g., to switch clients between multiple service providers at runtime transparently for load balancing issues, or to offer dependable services by using multiple service instances in parallel.

A third area for future work is the extension of the Smart Peer Group model toward Smart Environments and their integration into hybrid systems. While our approach so far allows an elemental integration of both system models by abstracting Smart Environment as very powerful peer devices, future system architectures could realize much more complex integrations.

Bibliography

[ACH$^+$01] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, and A. Hopper. Implementing a sentient computing system. *IEEE Computer Magazine*, 34(8):50–56, August 2001.

[AKUMK04] Jamal N. Al-Karaki, Raza Ul-Mustafa, and Ahmed E. Kamal. Data aggregation in wireless sensor networks – exact and approximate algorithms. In *Proceedings of IEEE Workshop on High Performance Switching and Routing (HPSR'04)*, April 2004.

[Atm06] Atmel. Data sheet ATmega8(L), revision Q. online publication, October 2006. http://www.atmel.com/dyn/resources/prod_documents/doc2486.pdf.

[AWSBL99] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP99)*. ACM, December 1999.

[BBHS03] Martin Bauer, Christian Becker, Jörg Hähner, and Gregor Schiele. ContextCube - providing context information ubiquitously. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003)*, May 2003.

[BCA$^+$01] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of Open ORB version 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.

[BCRP00] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of Middleware 2000*, 2000.

[BdM00] Luca Benini and Giovanni de Micheli. System-level power optimization: techniques and tools. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):115–192, 2000.

[BG00] Christian Becker and Kurt Geihs. Generic QoS-support for CORBA. In *Proceedings of 5th IEEE Symposium on Computers and Communications (ISCC'2000)*, 2000.

[BHS98] Luca Benini, Robin Hodgson, and Polly Siegel. System-level power estimation and optimization. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design (ISLPED'98)*, pages 173 – 178, August 1998.

[BHS04] Christian Becker, Marcus Handte, and Gregor Schiele. PCOM – a component system for pervasive computing. In *Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications (PerCom 04)*, Orlando, FL, USA, March 2004.

[BHSM04] Christian Becker, Marcus Handte, Gregor Schiele, and Pedro J. Marron. 3PC - peer to peer pervasive computing. In *GI FG-Treffen Betriebssysteme*, 2004.

[BKL01] Prithwish Basu, Naved Khan, and Thomas D.C. Little. A mobility based metric for clustering in mobile ad hoc networks. In *Proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW'01)*, April 2001.

[Blu04] Bluetooth Special Interest Group. Bluetooth core specification v2.0 + EDR. online publication, November 2004. http://www.bluetooth.com/Bluetooth/Learn/Technology/Specifications.

[BS03] Christian Becker and Gregor Schiele. Middleware and application adaptation requirements and their support in pervasive computing. In *Proceedings of the 23rd International Conference on Distributed Computing Systems Workshops (ICDCS 2003), 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems (DARES)*, pages 98–103, May 2003.

[BSGR03] Christian Becker, Gregor Schiele, Holger Gubbels, and Kurt Rothermel. Base – a micro-broker-based middleware for pervasive computing. In *Proceedings of the IEEE international conference on Pervasive Computing and Communications (PerCom)*, March 2003.

[CCCG97] Winston Liu Ching-Chuan Chiang, Hsiao-Kuang Wu and Mario Gerla. Routing in clustered multihop mobile wireless networks with fading channel. In *Proceedings of the IEEE Singapore International Conference on Networks (SICON'97)*, pages 197–211, April 1997.

[CJBM02] Benjie Chen, Kyle Jamieson, Hari Balakrishnan, and Robert Morris. Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. *ACM Wireless Networks Journal*, 8(5), September 2002.

[CMRW06] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language. W3C Candidate Recommendation, March 2006. http://www.w3.org/TR/wsdl20/.

[CPW$^+$99] M.H. Coen, B. Phillips, N. Warshawsky, L. Wiesman, S. Peters, and P. Finin. Meeting the computational needs of intelligent environments: The metaglue system. In *Proceedings of the 1st International Workshop Managing Interactions in Smart Environments (MANSE'99)*, pages 201–212, December 1999.

[CSA99] Jyh Cheng Chen, Krishna M. Sivalingam, and Prathima Agrawal. Performance comparison of battery power consumption in wireless multiple access protocols. *Wireless Networks*, 5(6):445–460, December 1999.

[CSC02] Inseok Choi, Hojun Shim, and Naehyuck Chang. Low-power color TFT LCD display for hand-held embedded systems. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 112–117, August 2002.

[CvHaDLM$^+$01] Dan Connolly, Frank van Harmelen, Ian Horrocks an Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. DAML+OIL (march 2001) reference description. W3C Note, December 2001. http://www.w3.org/TR/daml+oil-reference.

[DS01] Steffen Deter and Karsten Sohr. Pini – a Jini-like plug&play technology for the KVM/CLDC. In *Proceedings of the International Workshop on Innovative Internet Computing Systems (I2CS 2001)*, pages 53–67, June 2001.

[EE98] G. Eddon and H. Eddon. *Inside Distributed Com.* Microsoft Press, February 1998.

[Ell99] Carla Schlatter Ellis. The case for higher level power management. In *Proceedings of HotOS99*, March 1999.

[EWB87] Anthony Ephremides, Jeffrey E. Wieselthier, and Dennis J. Baker. A design concept for reliable mobile radio networks with frequency hopping signaling. *Proceedings of the IEEE*, 75(1):56–73, January 1987.

[Fer03] Reginald Ferber. *Information Retrieval – Suchmodelle und Data-Mining-Verfahren für Textsammlungen und das Web.* dpunkt.verlag, Heidelberg, Germany, 2003.

[Fli01] Jason Flinn. Extending mobile computer battery life through energy-aware adaptation. PhD thesis, TR No. CMU-CS-01-171, December 2001.

[FN01] L. M. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, Anchorage, AK, USA, April 2001.

[FS99] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Symposium on Operating Systems Principles*, pages 48–63, 1999.

[GB01] Kurt Geihs and Christian Becker. A framework for re-use and maintenance of quality of service mechanisms in distributed object systems. In *Proceedings of the IEEE International Conference on Software Maintenance*, 2001.

[GBS$^+$95] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. In *Winter USENIX Technical Conference*, pages 201–212, January 1995.

[GDL$^+$01] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Programming for pervasive computing environments. technical report UW-CSE-01-06-01, University of Washington, June 2001. submitted for publication.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco CA, USA, 1979.

[GPVD99] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC2608, June 1999.

[GSSS02] David Garlan, Daniel P. Siewiorek, Asim Smailagic, and Peter Steenkiste. Project aura: Toward distraction-free pervasive computing. *IEEE Pervasive Computing Magazine*, 1(2):22–31, April–June 2002.

[GT95] Mario Gerla and Jack Tzu-Chieh Tsai. Multicluster, mobile, multimedia radio network. *Wireless Networks*, 1(3), September 1995.

[HBS03] Marcus Handte, Christian Becker, and Gregor Schiele. Experiences - extensibility and minimalism in BASE. In *Proceedings of the Workshop on System Support for Ubiquitous Computing (UbiSys) at Ubicomp*, 2003.

[HCB02] Wendi B. Heinzelman, Anantha P. Chandrakasan, and Hari Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, October 2002.

[HGPC99] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hoc wireless networks. In *MSWiM '99: Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60, New York, NY, USA, August 1999. ACM Press.

[HHM$^+$00] Reto Hermann, Dirk Husemann, Michael Moser, Michael Nidd, Christian Rohner, and Andreas Schade. DEAPspace: transient ad-hoc networking of pervasive devices. In *Proceedings of the First Annual Workshop on Mobile Ad Hoc Networking & Computing (MobiHoc 2000)*, pages 133–134, August 2000.

[HHSB07] Marcus Handte, Klaus Herrmann, Gregor Schiele, and Christian Becker. Supporting pluggable configuration algorithms in pcom. In *Proceedings of the Workshop on Middleware Support for Pervasive Computing (PERWARE), International Conference on Pervasive Computing and Communications (PERCOM), to appear*, March 2007.

[HLMW02] Polly Huang, Vincent Lenders, Philipp Minnig, and Mario Widmer. Mini: A minimal platform comparable to Jini for Ubiquitous Computing. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA 02)*, October 2002.

[HMTR04] Daniel Herrscher, Steffen Maier, Jing Tian, and Kurt Rothermel. A Novel Approach to Evaluating Implementations of Location-Based Software. In *Proceedings of the 2004 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2004)*, pages 484–490, San Jose, CA, USA, July 25–29 2004.

[HSUB05] Marcus Handte, Gregor Schiele, Stephan Urbanski, and Christian Becker. Adaptation support for stateful components in PCOM. In *Proceedings of Pervasive 2005, Workshop on Software Architectures for Self-Organization: Beyond Ad-Hoc Networking*, 2005.

[IEE99] IEEE. IEEE 802.11 Standard: Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. Specification, 1999.

[JFW02] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing Magazine*, 1(2):71–78, April–June 2002.

[JSAC01] Christine E. Jones, Krishna M. Sivalingam, Prathima Agrawal, and Jyh Cheng Chen. A survey of energy efficent network protocols for wireless networks. *Wireless Networks*, 7(4):343–358, August 2001.

[JTK97] A.D. Joseph, J.A. Tauber, and M.F. Kaashoek. Mobile computing with the rover toolkit. *IEEE Transactions on Computers: Special issue on Mobile Computing*, 46(3):337–352, March 1997.

[KOA^{+}99] Cory D. Kidd, Robert J. Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth Mynatt, Thad E. Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proceedings of the Second International Workshop on Cooperative Buildings (CoBuild'99)*, October 1999.

[LCS^{+}00] Yung-Hsiang Lu, Eui-Young Chung, Tajana Simunic, Luca Benini, and Giovanni De Micheli. Quantitative comparison of power management algorithms. In *Design Automation and Test in Europe (DATE)*, pages 20–26, March 2000.

[Led99] Thomas Ledoux. OpenCorba: A reflective open broker. In *Proceedings of the 2nd International Conference on Reflection (Reflection'99)*, pages 197–214, 1999.

[Loo01] Don Loomis. *The TINI Specification and Developer's Guide*. Addison-Wesley, June 2001.

[LS98] Jacob R. Lorch and Alan Jay Smith. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.

[LSM99] Yung-Hsiang Lu, Tajana Simunic, and Giovanni De Micheli. Software controlled power management. In *7th International Workshop on Hardware/Software Codesign (CODES99)*, pages 157–161, May 1999.

[Man97] Steve Mann. Smart clothing: The wearable computer and wearcam. *Personal Technologies*, March 1997. Volume 1, Issue 1.

[McK00] D. McKenna. A methodology for evaluating mobile computing devices. Technical report, Transmeta Corporation, February 2000.

[Mic00] Microsoft Corporation. Universal plug and play device architecture, version 1.0. online publication, June 2000. http://www.upnp.org/download/UPnPDA10_20000613.htm.

[MNTW01] B. A. Miller, T. Nixon, C. Tai, and M. D. Wood. Home networking with universal plug and play. *IEEE Communications Magazine*, 39(12), December 2001.

[Moz98] Michael C. Mozer. The neural network house: An environment that adapts to its inhabitants. In *Proceedings of the American Association for Artificial Intelligence Spring Symposium on Intelligent Environments*, pages 110–114, March 1998.

[Nid00] Michael Nidd. Reducing power use in DEAPSpace service discovery. Research Report RZ 3254, IBM Research Laboratory Zürich, March 2000.

[Nid01] Michael Nidd. Service discovery in DEAPspace. *IEEE Personal Communications*, 8(4):39–45, August 2001.

[Nin06] Nintendo. Product information page for the nintendo ds game console. online publication, 2006. http://www.nintendo-europe.com/NOE/de/DE/system/nds_topic1.jsp.

[Obj98] Object Management Group. Object Management Group, CORBA messaging, report orbos/98-05-06. online publication, May 1998. http://www.omg.org/.

[Obj02] Object Management Group. Minimum CORBA specification, revision 1.0, August 2002.

[Obj04] Object Management Group. The common object request broker: Architecture and specification, revision 3.0.3. online publication, March 2004. http://www.omg.org/.

[PJKF03] Shankar R. Ponnekanti, Brad Johanson, Emre Kiciman, and Armando Fox. Portability, extensibility and robustness in iros. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications (PERCOM 2003)*, March 2003.

[PR00] Arno Puder and Kay Roemer. *MICO: An Open Source CORBA Implementation*. Morgan Kaufmann Publishers, 3rd edition edition, March 2000.

[QP99] Qinru Qiu and Massoud Pedram. Dynamic power management based on continuous-time markov decision processes. In *36th ACM/IEEE conference on Design automation*, pages 555–561, June 1999.

[RC00] Manuel Román and Roy H. Campbell. GAIA: Enabling active spaces. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[RC01] Manuel Román and Roy H. Campbell. Unified object bus: Providing support for dynamic management of heterogeneous components. Technical Report UIUCDCS-R-2001-2222 UILU-ENG-2001-1729, Universiy of Illinois at Urbana-Champaign, 2001.

[RJO+89] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: A system software kernel. In *Proceedings of the 34th Computer Society International Conference (COMPCON 89)*, February 1989.

[RKC99] Manuel Román, Fabio Kon, and Roy H. Campbell. Design and implementation of runtime reflection in communication middleware: The dynamictao case. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshops, Workshop on Electronic Commerce and Web-Based Applications*, pages 122–127, May 1999.

[RKC01] Manuel Román, Fabio Kon, and Roy H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, July 2001.

[RSC+99] Manuel Román, A. Singhai, D. Carvalho, C. Hess, and R.H. Campbell. Integrating PDAs into distributed systems: 2K and PalmORB. In *Proceedings of the International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, September 1999.

[SBG99] Albrecht Schmidt, Michael Beigl, and Hans-W. Gellersen. There is more to context than location. *Computers and Graphics*, 23(6):893–901, 1999.

[SBM99] Tajana Simunic, Luca Benini, and Giovanni De Micheli. Event-driven power management of portable systems. In *12th International Symposium on System Synthesis (ISSS)*, pages 18–23, November 1999.

[SBR04] Gregor Schiele, Christian Becker, and Kurt Rothermel. Energy-efficient cluster-based service discovery. In *Proceedings of the 11th ACM SIGOPS European Workshop*, September 2004.

[SBS02] Eugene Shih, Paramvir Bahl, and Michael J. Sinclair. Wake on wireless: An event driven energy saving strategy for battery operated devices. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom02)*, pages 160–171, Atlanta, GA, USA, September 2002.

[Sch04] Douglas C. Schmidt. Minimum TAO. online publication, June 2004. http://www.cs.wustl.edu/ schmidt/ACE_wrappers/TAO/docs/minimumTAO.html.

[SCI$^+$01] Eugene Shih, Seong-Hwan Cho, Nathan Ickes, Rex Min, Amit Sinha, Alice Wang, and Anantha Chandrakasan. Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom01)*, pages 272–287, Rome, Italy, July 2001.

[SCS00] Krishna M. Sivalingam, Jyh Cheng Chen, and Mani B. Srivastava. Design and analysis of low-power access protocols for wireless and mobile ATM networks. *Wireless Networks*, 6(1):73–87, January 2000.

[SK97] M. Stemm and R.H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Science, Special Issue on Mobile Computing*, 80(8):1125–1131, August 1997.

[Slo06] Slovak University of Technology. Intelligent pen: Handwriting on virtual paper. online publication, 2006. http://csidc.fiit.stuba.sk/2003/index-en.html.

[Soi00] Jonne Soininen. Gprs and umts release 2000 all-ip option. *ACM SIGMOBILE Mobile Computing and Communications Review*, 4(3):30–37, July 2000.

[SPP$^+$03] Umar Saif, Hubert Pham, Justin Mazzola Paluska, Jason Waterman, Chris Terman, and Steve Ward. A case for goal-oriented programming semantics. In *System Support for Ubiquitous Computing Workshop at the Fifth Annual Conference on Ubiquitous Computing (UbiComp03)*, Seattle, WA, USA, October 2003.

[SR98] S. Singh and C.S. Raghavendra. PAMAS: Power aware multi-access protocol with signalling for ad hoc networks. *Computer Communications Review*, 28(3):5–26, July 1998.

[Ste01] Martha Steenstrup. Cluster-based networks. In Charles E. Perkins, editor, *Ad Hoc Networking*, pages 75–138. Addison-Wesley, 2001.

[Sto04] Andreas Stoerzbach. *Emulation mobiler Geraete: Integration eines Batteriemodells*. Studienarbeit: Universität Stuttgart, Institut für Parallele und Verteilte Systeme, Verteilte Systeme, Januar 2004.

[Sun] Sun Microsystems. Java 2 platform, micro edition. http://java.sun.com/j2me.

[Sun01] Sun Microsystems. Jini technology core platform specification, version 1.2. online publication, December 2001.

[Sun02] Sun Microsystems. Java remote method invocation specification, revision 1.8. online publication, 2002. http://java.sun.com/j2se/1.4/docs/guide/rmi/index.html.

[Sun06] Sun Microsystems. Jini technology surrogate architecture specification, v1.0. online publication, 2006. http://surrogate.dev.java.net/specs.html.

[Sys06] Systronix. JStamp product homepage. online publication, 2006. http://jstamp.systronix.com/index.htm.

[THH02] Yu-Chee Tseng, Chih-Shun Hsu, and Ten-Yueng Hsieh. Power-saving protocols for IEEE 802.11-based multi-hop ad hoc networks. In *21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, volume 1, pages 200–209. IEEE, June 2002.

[TKvRB91] A.S. Tanenbaum, M.F. Kaashoek, R. van Renesse, and H. Bal. The amoeba distributed operating system-a status report. *Computer Communications*, 14(6):324–335, July/August 1991.

[TMWTC96] V. Tiwari, S. Malik, A. Wolfe, and L.M. Tien-Chien. Instruction level power analysis and optimization of software. *Journal of VLSI Signal Processing*, 13(2):1–18, August 1996.

[udd04] uddi.org. UDDI spec technical committee draft, version 3.0.2. online publication, October 2004. http://uddi.org/pubs/uddi_v3.htm.

[UHSR06] Stephan Urbanski, Marcus Handte, Gregor Schiele, and Kurt Rothermel. Experience using processes for pervasive applications. In *Proceedings of the Pervasive University Workshop at Informatik 2006*, 2006.

[Uni05] University of Bonn. BonnMotion – a mobility scenario generation and analysis tool. online publication, 2005. http://web.informatik.uni-bonn.de/IV/Mitarbeiter/dewaal/BonnMotion/.

[Wei91] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, September 1991.

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[WKY04] Chin-Hsien Wu, Tei-Wei Kuo, and Chia-Li Yang. Energy-efficient flash-memory storage systems with an interrupt-emulation mechanism. In *Proceedings of the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 134–139, September 2004.

[WL02] Karen H. Wang and Baochun Li. Group mobility and partition prediction in wireless ad-hoc networks. In *Proceedings of the IEEE International Conference on Communications (ICC)*, Apr 2002.

[WSR98] M. Woo, S. Singh, and C.S. Raghavendra. Power aware routing in mobile ad hoc networks. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom98)*, pages 181–190, October 1998.

[WWDS94] Mark Weiser, Brent Welch, Alan J. Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, pages 13–23, 1994.

[XBM$^+$03] Ya Xu, Solomon Bien, Yutaka Mori, John Heidemann, and Deborah Estrin. Topology control protocols to conserve energy in wireless ad hoc networks. Technical Report 6, University of California, Los Angeles, Center for Embedded Networked Computing, January 2003.

[XHE01] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (Mobicom01)*, pages 70–84, Rome, Italy, July 2001. USC/Information Sciences Institute, ACM.

[XLWN03] Rong Xu, Zhiyuan Li, Cheng Wang, and Peifeng Ni. Impact of data compression on energy consumption of wireless-networked handheld devices. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS03)*, pages 302–311, May 2003.

[Xu02] Ya Xu. *Adaptive energy conservation protocols for wireless ad hoc routing*. PhD thesis, University of Southern California, 2002. Adviser-John Heidemann and Adviser-Deborah Estrin.

[YC05] Jane Y. Yu and Peter H.J. Chong. A survey of clustering schemes for mobile ad hoc networks. *IEEE Communications Surveys and Tutorials*, 7(1):32–48, January 2005.

[YHE02] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM02)*, June 2002.

[ZELV03] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: A unifying abstraction for expressing energy. In *Proceedings of the Usenix Annual Technical Conference (Usenix03)*, pages 43–56, San Antonio, TX, USA, June 2003.

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(Gregor Schiele)