Middleware for service provision in disconnected mobile ad hoc networks

Romeo Said

▶ **To cite this version:**

Romeo Said. Middleware for service provision in disconnected mobile ad hoc networks. Networking and Internet Architecture [cs.NI]. Université de Bretagne Sud, 2011. English. NNT : . tel-00596996

HAL Id: tel-00596996

https://theses.hal.science/tel-00596996

Submitted on 30 May 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE / UNIVERSITÉ DE BRETAGNE SUD

UFR Sciences et Sciences de l'Ingénieur

sous le sceau de l'Université Européenne de Bretagne

Pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ DE BRETAGNE SUD

Mention : Informatique

École Doctorale SICMA

présentée par

Romeo SAID

VALORIA Laboratoire de Recherche Informatique et ses Applications de Vannes et Lorient

Middleware for service provision in disconnected mobile ad hoc networks

Intergiciel pour la fourniture de services dans les réseaux mobiles ad hoc discontinus

Thèse soutenue le 23 février 2011,
devant la commission d'examen composée de :

M. Pierre-François Marteau
Professeur des Universités, Université de Bretagne-Sud / Président

Mme. Françoise André
Professeur des Universités, Université de Rennes 1 / Rapporteur

M. Didier Donsez
Professeur des Universités, Université de Grenoble 1 / Rapporteur

M. Thomas Ledoux
Maître-assistant, École des Mines de Nantes / Examinateur

M. Frédéric GUIDEC
Maître de Conferences HDR, Université de Bretagne-Sud / Directeur de thèse

M. Yves MAHÉO
Maître de Conferences, Université de Bretagne-Sud / Encadrant de thèse

"Everything should be made as simple as possible, but not simpler."

Albert Einstein

Acknowledgments

I would like to thank my supervisor Yves Mahéo for all his support. With his critical thinking and patience, he helped this work mature. I also want to thank my thesis director Frédéric Guidec for his guidance. I specifically thank him for reviewing this work, which greatly helped produce a better quality manuscript.

I thank the jury members, Françoise André and Didier Donsez for reviewing this manuscript, Thomas Ledoux as examiner, and Pierre-François Marteau as jury's president.

I thank all my colleagues at the VALORIA lab and all my friends, they all helped me go through this milestone.

Last but not least, I thank my family for their everlasting support. I specially dedicate this thesis to my loving parents.

This work was supported by the French *Agence Nationale de la Recherche* under contract ANR-05-SSIA-0002-01, by the *Région de Bretagne*, and by the *Conseil Général du Morbihan*.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Mobile ad hoc networks (MANETs) . . . 2

1.1.2 The disconnected reality of MANETs . . . 2

1.1.3 Need for application support . . . 3

1.1.4 Service-oriented computing: advantages and challenges . . . 4

1.2 Goal . . . 5

1.3 Organization of the manuscript . . . 5

2 Service-Oriented Computing and Disconnected Mobile Ad Hoc Networks **7**

2.1 Introduction . . . 7

2.2 Communications in disconnected mobile ad hoc networks . . . 8

2.2.1 Communication protocols for MANETs . . . 8

2.2.1.1 Proactive routing . . . 9

2.2.1.2 Reactive routing . . . 9

2.2.2 Communication protocols for disconnected MANETs . . . 9

2.2.2.1 Delay tolerant networking . . . 10

2.2.2.2 Message dissemination protocols . . . 10

2.2.2.3 Content-based communications . . . 11

2.3 Service-oriented computing . . . 11

2.3.1 From object-oriented to component-oriented distributed systems . . . 12

2.3.2 Service-oriented paradigm . . . 12

2.3.3 Fundamental steps of the service provision process . . . 13

2.3.4 Client/provider interactions in distributed computing . . . 14

2.3.4.1 Message passing . . . 15

2.3.4.2 Remote procedure call . . . 15

2.3.4.3 Event notifications . . . 15

2.3.4.4 Tuple spaces . . . 16

2.3.4.5 Message queues . . . 16

2.3.4.6 Publish/Subscribe . . . 16

2.4 Service provision systems . . . 17

2.4.1 Service provision systems for stable networks . . . 17

2.4.1.1 OSGi . . . 17

2.4.1.2 Enterprise services . . . 18

2.4.1.3 Web Services 18
2.4.1.4 Service Component Architecture 20
2.4.2 Service provision systems in dynamic networks 20
2.4.3 Service provision systems in MANETs 22
2.4.4 Discussion 24
2.5 Conclusion 26

3 Challenges and Design Overview **29**
3.1 Introduction 29
3.2 Challenging points 30
3.2.1 Communication 30
3.2.2 Interoperability 30
3.2.3 Service contract 31
3.3 Design overview 31
3.4 Summary 32

4 Disconnected Communication Support **33**
4.1 Introduction 33
4.2 DoDWAN communication protocol 34
4.2.1 Opportunistic gossiping 34
4.2.2 Periodic announcements 35
4.2.3 Local cache 35
4.2.4 Content-based matching 35
4.2.5 Mobility as an advantage 35
4.2.6 k-hop opportunistic gossiping 36
4.2.7 Frugal use of the wireless medium 36
4.3 Publish/Subscribe interface 36
4.3.1 Message 36
4.3.2 Publishing a message 37
4.3.3 Subscribing to messages 38
4.3.4 Canceling a message 39
4.4 Discussion 39
4.4.1 Communication delay 39
4.4.2 Loose coupling benefits 40
4.5 Summary 40

5 Service Discovery **41**
5.1 Introduction . . . 41
5.2 Service nodes in mobile environments . . . 42
5.3 Overview . . . 44
5.4 Elements of the discovery protocol . . . 46
5.4.1 Description . . . 46
5.4.1.1 Functional and non-functional service properties . . . 47
5.4.1.2 Provider context properties . . . 47
5.4.2 Advertisement . . . 48
5.4.3 Process at the provider side . . . 52
5.4.4 Collection . . . 53
5.4.5 Selection . . . 53
5.4.6 Process at the client side . . . 54
5.5 Discussion . . . 54
5.6 Summary . . . 55

6 Service Invocation Solutions **57**
6.1 Introduction . . . 57
6.2 Service provider redundancy . . . 58
6.3 Remote invocation . . . 60
6.3.1 How clients issue invocation requests to providers . . . 60
6.3.2 How providers respond to client requests . . . 62
6.3.3 Response management policy . . . 66
6.3.4 Network healing . . . 67
6.3.4.1 Reactive competition . . . 68
6.3.4.2 Safe healing . . . 68
6.3.4.3 Aggressive healing . . . 69
6.3.5 Discussion . . . 71
6.3.5.1 Client-Provider binding . . . 71
6.3.5.2 Loose coupling benefits . . . 72
6.4 Invocation restrictions . . . 73
6.5 Blind invocation: Bypassing discovery . . . 75
6.6 Client and provider states . . . 76
6.7 Remote invocations of stateful services . . . 77
6.8 Public invocations . . . 80
6.9 Perspectives . . . 81

6.9.1 Semantic invocation 81
6.9.2 Complex request 81
6.10 Summary 82

7 Implementation and Evaluation **83**
7.1 Introduction 83
7.2 The service platform's middleware 83
7.3 Simulation environment and evaluation metrics 86
7.4 Discovery evaluation 87
7.5 Invocation evaluation 89
7.5.1 Effects of invoking multiple providers 89
7.5.2 Network healing 91
7.5.3 Session recovery for stateful invocations 93
7.6 Conclusion 94

8 Conclusions and Perspectives **97**
8.1 Conclusions 97
8.2 Perspectives 99

References **104**

List of Figures

1.1 Communications in a Mobile Ad Hoc Network (MANET) 2
1.2 Disconnected Mobile Ad Hoc Network 3
1.3 Middleware layers 5

2.1 Entity interactions in a service oriented architecture 13

3.1 Two layer service platform on each mobile network node 32

4.1 Message examples 37
4.2 Publish/Subscribe and subsequent communications 38
4.3 Subscription elements 39
4.4 Predicate example 39

5.1 Lifecycle of a provider node and of a provider agent 43
5.2 Lifecycle of a client node and of a client agent 43
5.3 Network environment 44
5.4 Service interactions between a provider node *A* and client nodes *B* and *C* . 45
5.5 Components of a service descriptor message 46
5.6 Example of functional service properties written in WSDL 47
5.7 Example of non-functional service properties written in XML 47
5.8 Example of a provider node's context properties 48
5.9 Controlling the level of accessible information 49
5.10 Header attributes of a descriptor message 49
5.11 Example of a descriptor message 51
5.12 Provider side process, from service creation to getting ready for invocation 52
5.13 Process at the client side 54

6.1 Provider examples offering the same business service 58
6.2 Invocation request's header attributes 61
6.3 Request message example 62
6.4 Invocation response's header attributes 63
6.5 Example response messages 65
6.6 Destination Vs content-based invocation 66
6.7 Client-initiated healing message example 69
6.8 Healing predicate corresponding to the client's healing message 69
6.9 Provider-initiated healing message examples 70

6.10 Healing predicate corresponding to P2's healing message 70
6.11 Stateless invocations, late binding compared to loose binding 71
6.12 Pushing loose coupling to the limits 73
6.13 Request messages for restricted invocations 73
6.14 Multi-level addressing in a request enables a wider impact range 74
6.15 Blind request message 75
6.16 Client and provider states 76
6.17 Stateful sequential invocation messages 79
6.18 Public invocation messages 80
6.19 A complex request example 82

7.1 Architecture of the service platform 84
7.2 Simulation environment of a disconnected MANET 86
7.3 Performance of discovery 88
7.4 Destination-based Vs content-based invocations 90
7.5 Discovery and invocation satisfactions for 10% provider replication 90
7.6 Performance of healing 92
7.7 Ratio of received responses over requests 92
7.8 Cumulative load sent into the network medium 93
7.9 Session management. 94

8.1 Quality of service adaptation persperctives 100

List of Tables

7.1 Response reduction 91

1 Introduction

Contents

1.1 Motivation **1**

1.1.1 Mobile ad hoc networks (MANETs) 2

1.1.2 The disconnected reality of MANETs 2

1.1.3 Need for application support 3

1.1.4 Service-oriented computing: advantages and challenges 4

1.2 Goal **5**

1.3 Organization of the manuscript **5**

1.1 Motivation

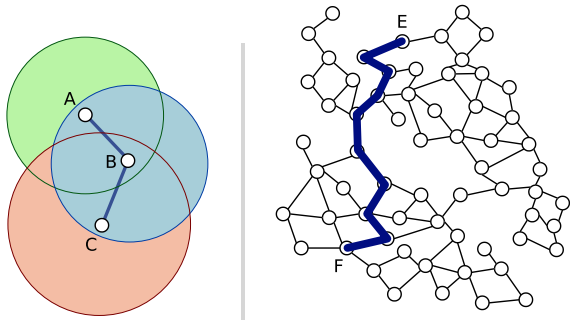
The term "ubiquitous computing" refers to the pervasiveness of digital systems having local processing power. The proliferation of computers gives birth to mobile devices with heterogeneous hardware resources like CPU, memory, and storage (from laptops, to personal digital assistants, to mobile phones, etc.). This proliferation also produces smart objects which are everyday objects with embedded processing power, enabling them to interact with the environment and to locally process the information. This representation is usually referred to as "pervasive computing."

Ubiquitous computing can also refer to the possibility of accessing processing power anywhere and at any point of time in order to satisfy any processing need. The focus is on connectivity, where a device is able to request some processing capabilities residing on another device. Mobile devices have heterogeneous communication resources, from accessing a wired network, to having wireless chipsets with long radio ranges for cellular networks, to having wireless chipsets with limited radio ranges (e.g. Wi-Fi, Bluetooth, or Zigbee).

For this work, I am interested in network environments with no fixed infrastructure, where only wireless ad hoc communications are possible between neighboring devices. In such environments, devices can offer their processing power as services for other devices to use.

1.1.1 Mobile ad hoc networks (MANETs)

Mobile ad hoc networks (MANETs) are spontaneously formed out of a number of mobile devices that communicate thanks to short-range wireless communication capabilities, using for example Wi-Fi or Bluetooth interfaces. A main advantage of this kind of network is that it can be used without deploying a specific and sometimes costly infrastructure (such as interconnected Wi-Fi access points). Two devices are able to communicate when they are in the wireless range of one another. Figure 1.1-(a) shows an



(a) Devices within wireless range are connected in ad hoc mode.

(b) Illustration of a fully connected MANET.

Figure 1.1: Communications in a Mobile Ad Hoc Network (MANET)

example in which devices B and C can communicate because they are in mutual wireless range. Devices A and C on the other hand cannot communicate directly but they can use B as an intermediate relay. Therefore, mobile devices can form a network, where each device represents a network node. Figure 1.1-(b) illustrates a typical MANET, where nodes are well distributed to enable a fully connected network. In this type of MANET, any two nodes like E and F can communicate using a route created through intermediate nodes. Because all nodes can move continuously (and often unpredictably) in such a network, routes must be created and maintained by dynamic routing protocols [1]. For example the route illustrated between nodes E and F can be dynamically reconstructed when the topology of the network changes. MANETs actually cover a wide variety of situations depending on the density of nodes in the network or their mobility scheme.

1.1.2 The disconnected reality of MANETs

In many ad hoc networks deployed in real conditions, mobile nodes can exhibit highly dynamic behaviors of mobility and volatility. Volatility refers to the fact that they may temporarily be switched off. Because of their mobility, their limited radio-range and their volatility, the devices in such network environments form so-called "islands" whose topology evolves continuously, rather than a single fully connected network, as shown in Figure 1.2. The fragmentation of the network into islands renders communication between two islands impossible. As a consequence, a temporaneous path cannot always be established between any pair of nodes in the network: end-to-end connectivity is thus not guaranteed. In this work, I focus on this class of MANETs which I refer to as *disconnected*

MANETs throughout the rest of this document. Network-wide communication itself in disconnected MANETs is still a challenge, namely because routing techniques designed for fully connected MANETs cannot be applied.

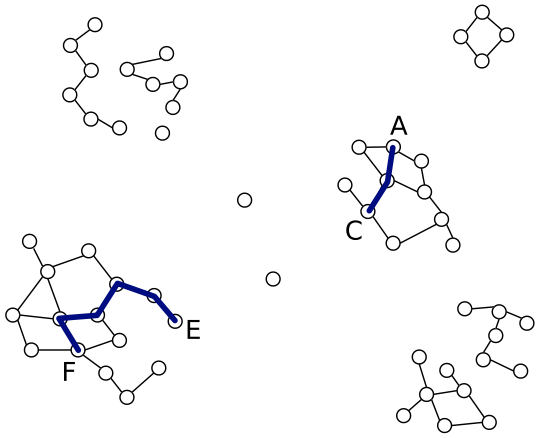


Figure 1.2: Disconnected Mobile Ad Hoc Network. Temporaneous communications between network islands are not possible.

Communication within an island remains possible, though. For example in the disconnected MANET shown in Figure 1.2, a route can be created between nodes A and C and between E and F, but there is no feasible route between C and E.

As far as communications in disconnected MANETs are concerned, the absence of end-to-end connectivity precludes relying on traditional routing techniques. Traditional routing algorithms strive to identify a succession of nodes that form a path between a sender and a receiver, in order to be able to transmit a message temporaneously along this path. When routes cannot be created, the *store-carry-and-forward* approach [2] provides a solution. With this approach, a message can be stored temporarily on a node, in order to be forwarded later when circumstances permit. Mobility then becomes an advantage as it facilitates message propagation: a device can carry a message when moving from an island to another. Messages are therefore forwarded from one network node to another when these nodes have the opportunity to meet.

1.1.3 Need for application support

MANETs have originally motivated research for tactical communication. Yet beside the military domain, MANET applications in the civil domain are also considered, such as: tourist services in infrastructure-less sites, traffic information in town centers, disaster relief situations, or social networks in university campuses or societies. Such applications need to be implemented using distributed application technologies.

When environments are modeled as fully connected MANETs, there is no special need for dedicated distributed application systems, since routing protocols are supposed to provide seamless network-wide connectivity. As a consequence, implementing distributed application technologies is not expected to be a challenge in such conditions.

When we recognize that real world MANETs are actually mostly disconnected, we realize that existing distributed application solutions are not appropriate. Effectively, these distributed application solutions are mostly designed for connected networks. Moreover, because of the absence of network wide end-to-end connectivity, these solutions get restricted to proximity interactions between a node and its neighboring nodes only. Therefore there is a need for building network-wide distributed application support that specifically target disconnected MANETs, and that can tolerate connectivity disruptions and transmission delays in such environments.

1.1.4 Service-oriented computing: advantages and challenges

Computing in mobile environments is increasingly becoming an experience where users interact with their mobile devices and need access to more and more distant services. In the service-oriented computing paradigm, the service provision process allows a user to exploit services offered by other devices in the network by discovering and using their capabilities. In comparison with object-oriented and component-oriented distributed computing, service-oriented computing offers the means to manage a high-level control over the provision process.

The three basic interacting entities in a service-oriented architecture are a client, a provider (or server), and a directory. Providers register their offered services in the directory. A client searches this directory for a needed service. It can then use the discovered service by remotely invoking the provider that is hosting it. Unlike traditional client/server invocation approaches, the service-oriented approach introduces late-binding between the client and the provider. The client invokes the most convenient provider at runtime so it is not bound to a single provider.

The service-oriented computing model seems suited for ad hoc environments because it emphasizes the decoupled nature of its entities. Decoupling between the client and the provider is essential in mobile environments with a fluctuating availability of providers, and where end-to-end communications are not guaranteed.

Still, in existing service-oriented systems, providers are usually supposed to be always available, as it is the case for example in Web services. Providers are also assumed to be continuously reachable in wireless mobile environments, where local connected networks can be created using Wi-Fi hotspots, or by creating ad hoc networks using proximity one-hop or routed multi-hop protocols. Therefore, although the service computing model is presented as loosely coupled, end-to-end invocation interactions between a client and a provider are not supposed to be really challenging.

Although the service-oriented approach seems relevant for disconnected MANETS, implementing distributed services for such networks still poses several challenges. Not only network-wide communication features must be provided, in spite of constant network fragmentation, but aspects such as the unpredictable reachability of the providers, or potential communication delays, must be taken into account at the service level. Migrating the service-oriented model from stable networks to disconnected mobile ad hoc networks therefore requires rethinking of many concepts.

1.2 Goal

This thesis investigates the benefits and challenges of using the service-oriented paradigm for mobile computing in disconnected mobile ad hoc network environments. My overall objective is to build a service middleware platform for mobile nodes that supports the execution of service-oriented applications in such environments.

In order to design a middleware platform that supports the execution of service-oriented applications, I consider the service-oriented computing approach and how communications can be provided in the targeted environments. I present the state of the art of both communications and service-oriented systems in mobile ad hoc networks. I also present the challenges that need to be addressed in order to design such a platform.

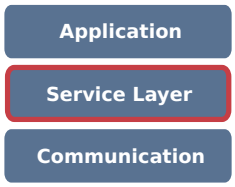


Figure 1.3: Middleware layers

I then propose a service platform implemented as a middleware composed of two layers: a communication layer, and a service layer (see Figure 1.3). The communication layer has to provide mechanisms to decouple two service entities in terms of temporaneous interaction, synchronous behavior, and mutual knowledge. For this layer, I use solutions for network-wide communication that were developed by my research team (CASA) at the Valoria laboratory. The service layer has to provide mechanisms that decouple service providers and clients in terms of interoperability and service contract. For this layer, I propose solutions for service discovery and invocation.

1.3 Organization of the manuscript

Chapter 2 presents the state of the art for service-oriented computing and for communications in mobile ad hoc networks. It starts by defining mobile ad hoc networks, then it presents the main communication protocols used in such networks. The chapter also details service-oriented computing by presenting the principles of the service-oriented design paradigm and the fundamental steps of the service provision process. And finally it presents existing service provision systems, detailing those designed for stable networks, more dynamic networks, and those designed for MANETs.

Chapter 3 introduces my contribution. It presents the challenging points that must be addressed when designing a service provision platform that remains viable in disconnected mobile environments. The aim is to achieve better entity independence and loose coupling at each of the presented points. The chapter also presents the design overview of the service provision platform consisting of a communication layer and a service layer.

Chapter 4 presents the communication layer of the service platform. Due to the high level of communication constraints imposed by disconnected environments, I used an

opportunistic and content-driven protocol (DoDWAN). The chapter describes the inner workings of the protocol implementing a store-carry-and-forward paradigm, opportunistic gossiping, and content-based matching. The chapter also describes the publish/subscribe module interfacing the protocol with the upper service layer.

Chapter 5 presents the discovery protocol of the service layer. It describes the client and provider nodes and their life-cycles, and describes the architecture of local directories. It details the elements that construct the discovery protocol, from the provider's service description and advertisement to the client's service collection and selection.

Chapter 6 presents the full invocation solutions of the service platform. Invocations must be tolerant to communication delays due to disconnected network environments. The chapter presents the redundancy of service providers in mobile environments, where there might be more than one provider offering the same business service. The chapter presents the default remote invocation solution, detailing how a client can issue a content-based invocation request to all providers of a business service, and how these providers can respond back, and how network healing techniques eliminate unneeded messages. It also presents how a client can restrict the invocation to specific providers. In case of a discovery failure, a client can even send a blind invocation request hoping for matching provider to respond. The chapter also presents stateful remote invocation solutions via session management.

Chapter 7 presents the middleware implementation of the service platform described in the previous chapters. The middleware's layout consists of a service layer called DiSWAN where service agents reside, that can access the DoDWAN communication layer via a publish/subscribe interface. Using this implementation, I conducted simulations in as realistic as possible conditions in order to evaluate the service platform. The main objective is to assess the performance of service discovery and invocation in terms of response time and network load.

Chapter 8 concludes this document and presents some perspectives for future work.

2

Service-Oriented Computing and Disconnected Mobile Ad Hoc Networks

Contents

2.1 Introduction 7
2.2 Communications in disconnected mobile ad hoc networks 8
2.2.1 Communication protocols for MANETs 8
2.2.2 Communication protocols for disconnected MANETs 9
2.3 Service-oriented computing 11
2.3.1 From object-oriented to component-oriented distributed systems 12
2.3.2 Service-oriented paradigm 12
2.3.3 Fundamental steps of the service provision process 13
2.3.4 Client/provider interactions in distributed computing 14
2.4 Service provision systems 17
2.4.1 Service provision systems for stable networks 17
2.4.2 Service provision systems in dynamic networks 20
2.4.3 Service provision systems in MANETs 22
2.4.4 Discussion 24
2.5 Conclusion 26

2.1 Introduction

This chapter presents the state of the art for service-oriented computing and communications in mobile ad hoc networks. Section 2.2 starts by defining mobile ad hoc networks, then it presents the main communication protocols used for these networks. In connected MANETs (Section 2.2.1), communications are implemented using dynamic routing protocols, where routing tables are created and maintained mainly using either proactive or reactive routing techniques. In disconnected MANETs (Section 2.2.2), the network contains disconnected islands between which routing protocols do not work. In these disconnected networks, communications must be tolerant to delays and disruptions, and protocols mostly use a store-carry-and-forward mechanism to enable network-wide communications. Protocols rely on message delivery using either custody transfer (allowing only one message copy) or epidemic behaviors (many message copies are disseminated), some protocols use special techniques to reduce the overhead of epidemic

dissemination. Protocols can support destination-based or content-based communication styles. Section 2.3 details service-oriented computing by presenting the principles of the service-oriented design paradigm (Section 2.3.2) and the fundamental steps of the service provision process (Section 2.3.3). It also presents the techniques and technologies that are used to enable interactions between clients and providers in distributed systems (Section 2.3.4). Section 2.4 presents the service provision systems. It details those designed for stable networks (mainly enterprise and web services) and more dynamic networks (e.g. LAN). Section 2.4 also presents provision systems designed for MANETs, and discusses their main characteristics. Section 2.5 concludes the chapter.

2.2 Communications in disconnected mobile ad hoc networks

Mobile networks are often thought of as mobile computers connected to the internet through cellular networks (e.g. EDGE, UMTS, HSPA [3]) or through Wi-Fi access points (IEEE 802.11 [4]). These networks are called infrastructure-based networks. MANETs are a completely different type of networks, where mobile computers can communicate with each other via short-range wireless standards (e.g. Wi-Fi [4] in ad hoc mode, Bluetooth IEEE 802.15 [5]). MANETs are spontaneously formed networks that do not need any form of infrastructure.

2.2.1 Communication protocols for MANETs

MANETs can be formed by pedestrians holding mobile computers like smartphones, PDAs, netbooks, or laptops. Another type are the VANETs [6] (Vehicular Ad Hoc Networks) where computers are embedded in vehicles. Another type of MANETs are sensor networks, where sensors are used to monitor wild-life animals [7, 8], or monitor temperatures and other physical parameters over wide areas. Sensors networks usually focus on very low energy consumption.

Mobile ad hoc networks were first introduced in the military domain, where the spontaneous network organization is considered as a big advantage in the battle field. But in recent years, an increased number of studies have brought these networks to the civil domain. One example is creating MANETs for disaster relief situations where all infrastructures could be destroyed, and where there is a need to coordinate rescue teams. Another example is the use MANETs to provide communications for developing nations [9]. MANETs can be an alternative to infrastructure-based networks, either because of high costs or simply enabling ad hoc social and community interactions.

Communications in MANETs are often based on IP routing protocols, where every network node is considered as a router, and where routing tables are created in order to enable end-to-end communications. IP packets can be sent from a source to a destination node through many intermediate router nodes. Networks should have high node density in order to support multi-hop connectivity between any two nodes. Routing protocols manage the routing table creation and updates in order to support node mobility. The IETF MANET working group is in the process of standardizing many routing protocols corresponding to two families: Proactive routing and reactive routing. Other protocols are considered as hybrid, they use both proactive and reactive techniques. Other

protocols use geographical locations at each node in order to build routing tables.

2.2.1.1 Proactive routing

Proactive or table-driven routing protocols require that each node of the network has to constantly keep its own routing table up-to-date and ready in case a route is needed. Each node periodically broadcasts its routing table to neighboring nodes. The main advantage of this type of routing, is that an end-to-end communication route can always be found. But the disadvantage is that control messages used to keep the tables up-to-date involves heavy network traffic, specifically if the network is not static (e.g. node mobility). The OLSRv2 [10] protocol is the reference proactive protocol.

2.2.1.2 Reactive routing

Reactive or on-demand routing protocols does not require to keep up-to-date tables at each node. The routing table is created at a node when a communication to a destination node is needed. The source node invites other network nodes to update their tables in order for the packet to be routed to the destination node. The main advantage is that network traffic is proportional to the actual amount of communicated data. But the disadvantage is the delay when trying to create a route for the first time, or when the network topology changes due to mobile nodes. The DYMO [11] protocol is the reference in reactive protocols.

2.2.2 Communication protocols for disconnected MANETs

The above-mentioned MANET routing protocols consider that an end-to-end route can be dynamically created using intermediate nodes between any two nodes of the network. Therefore, intermediate nodes must be available and well distributed in space, in order for the communication to succeed. In case one intermediate node becomes unaccessible during a communication, the communication fails. However in many real-world situations, the network does not have enough node density to guarantee end-to-end connections between any two nodes. This type of network is called a disconnected MANET. Disconnected MANETs are characterized by the fragmentation of the network into isolated connectivity islands, where routing works inside of an island but not between islands, and therefore traditional routing is no longer appropriate for this kind of network. Moreover, connectivity can become even worse if we take into account the volatility of nodes (e.g. a node can be turned of, or be on standby).

In this work, I am interested in disconnected MANET environments, where network-wide communications is still considered a challenge due to the lack of temporaneous end-to-end connectivity. Communication protocols in disconnected MANETs must be disruption-tolerant and use a store-carry-and-forward paradigm. They mainly enable relay network nodes to store and carry data from one network island to another. Disconnected MANET protocols can be destination-based or content-based. Destination-based protocols present a send/receive like interface: a message is sent to one or many specifically addressed destination nodes, and a node receives only messages that are specifically

destined to it. Content-based protocols present a publish/subscribe like interface: a message is published with no specific destination, and a node subscribes for messages of interesting content in order to receive it.

2.2.2.1 Delay tolerant networking

Communications in disconnected MANETs are getting more and more attention in recent years. Protocols try to enhance traditional routing for connected MANETs by tolerating disconnections. These enhancements consist of enabling some or all of the network nodes to temporarily store messages, in order to resend them later on when conditions permit. This type of networking is called DTN for Delay Tolerant Networking, and is federated by the DTNRG working group [2] of the Internet Research Task Force. DTN architectures were originally focused on interplanetary networks where communication delays occur because of the long distances between communicating parties, and because of their positioning in space (direct link availability). DTNRG also study tactical military environments and other forms of disconnected environments. DTN protocols rely on the store-carry-and-forward mechanism.

The store-carry-and-forward approach enables network nodes to temporarily store a message (in a local cache) if it is incapable of immediately transferring the message to another network node. Therefore nodes can carry messages around, and due to their mobility, they can transfer these messages to other disconnected parts of the network environment. Messages can be transferred using dynamic routing protocols inside of connectivity islands, but they should be carried by the moving nodes between disconnected islands.

2.2.2.2 Message dissemination protocols

From the store-carry-and-forward approach, came the idea of opportunistic networking in dynamic and disconnected MANETs. In opportunistic networking, a node takes the opportunity of meeting another node in order to exchange messages. Opportunistic networking do not rely on routing tables, messages are transfered from node to node, and contacts between nodes are not planed. There exists many opportunistic DTN networking studies, [12, 13, 14] are surveys of the proposed solutions. Communication protocols vary depending on the characteristics of the target environment like the network size and node mobility patterns, and one-to-one or one-to-many communication needs.

As an example of one-to-one communications, a message can be routed from a source node to a destination node using custody transfer [15]. A single copy of the message is transferred from node to node. Amongst possible contacts, a node delegates the message to the best possible carrier node. This is only convenient for environments where node mobility is well known or highly predictable. Custody transfer has the advantage of producing very low network traffic, but communication failures occur in case a delegated carrier node fails.

One-to-many communications are usually implemented using epidemic dissemination protocols [16, 17]. Message copies propagate like a virus from one network node to every other node it meets during its travels, and each infected node does the same,

until the message reaches all network nodes. The advantage of this behavior is that a message gets transferred to its destinations very fast, and the communication does not fail in case some of the carrier nodes fail. But the disadvantage is that it creates a high network traffic and consumes storage space at each node. Some protocols [18, 19] try to minimize these disadvantages, by stopping the dissemination after the message gets delivered to destination, or by limiting the propagation of the dissemination by fixing a message lifetime or using geolocation techniques. Others control the dissemination of a message, where each node selects the best possible carriers using a utility function. This utility function is usually based on studying the history of contacts and mobility patterns in order to predict the future [20], or the utility function can use probabilistic methods [21].

2.2.2.3 Content-based communications

The majority of routing protocols are destination-based, where the sender of a message explicitly specifies the addresses of one or many destination nodes. The message gets routed in the network in order to be received only by the specified destinations. Another type of communication protocols are the content-based protocols (e.g. [22, 23]). In content-based communications, a message does not contain an explicit destination. The message gets routed in the network and delivered according to the interest that network nodes have for the contents of the message. Therefore, the communication behavior is similar to the content-based publish/subscribe model [24]. A source node publishes messages in the network, other nodes subscribe to some interesting messages, and the routing protocol brokers the delivery of these messages.

The DoDWAN protocol [25, 26] is a protocol that builds on the previously mentioned approaches to enable communications in highly dynamic disconnected MANETs. It uses the store-carry-and-forward mechanism to opportunistically disseminate messages in the network, and relies on the content-based approach for the opportunistic message exchange. Compared to other previously mentioned protocols, the DoDWAN protocol enhances message delivery in highly dynamic disconnected MANETs while producing a fairly low network traffic. It provides mechanisms to decouple two service entities in terms of temporaneous interaction, synchronous behavior, and mutual knowledge. Therefore, I chose this protocol for the communication layer of the service platform. Chapter 4 details how this protocol provides communication support in the service platform.

2.3 Service-oriented computing

A service is a task or a function that a provider can provide to a consumer. Service-orientation is a design paradigm that helps build distributed computing systems around the service as a fundamental unit [27]. Service-oriented computing systems must follow the principles of the service-oriented design paradigm. The service-oriented paradigm gets its inspiration from object-oriented computing principles, it also shares some goals with component-oriented computing, but offers further enhancements and additions.

2.3.1 From object-oriented to component-oriented distributed systems

A distributed system consists of multiple autonomous computers that communicate through a computer network. The computers interact with each other in order to achieve a common goal [28]. Computers are autonomous entities, each having a processor and a local memory, and they communicate with each other by message passing. More explicitly, a computer is represented by a software agent, an agent communicates by protocol stacks over a network with remote agents. Agents work together to perform some tasks, in a distributed manner.

Remote Procedure Call RPC [29] was first introduced in procedural languages. Since then, the main distributed systems are object-oriented and component-oriented distributed systems.

In distributed object systems, a software agent exposes the semantics of object initialization and method invocation to remote agents. Objects maintain complex internal states to support their methods. Proprietary or standardized mechanisms broker communications across system boundaries. Communication interactions between remote agents are fine grained, and these agents must share the same implementation technology. One object sends a request to another object in a remote machine to perform a task, the result is sent back to the calling object. Examples of technologies that implement distributed objects are: Java RMI [30], DCOM [31], Objective-C [32], Pyro [33]. The CORBA [34] technology enhances distributed object systems by allowing agents to have different implementation technologies.

In distributed component systems, a software component is an encapsulated entity that communicates with other components via interfaces. A component offers its services through a *provided* interface, another component needing these services adopts a *used* interface. This is how components communicate without being concerned with the inner workings and implementations of one another. A computer running several components is called an application server, the combination of application servers and software components invoking them forms a distributed system. Examples of distributed software component technologies are: CORBA Component Model [34], Enterprise Java Beans [35], .NET Remoting [36], FractalRMI and Dream [37].

2.3.2 Service-oriented paradigm

The service consists of a set of capabilities and a contract describing them to the public. The contract must be standardized so that consumers properly understand the capabilities offered by providers. The contract expresses the functional aspect of the service (functions and data types), as well as the policy aspect of the service (non-functional and behavioral characteristics).

Services must be discoverable and easily located by consumers who want to interoperate with them. Discovery mechanisms involve using a service registry (or directory) containing service contracts, that a consumer can access to choose the needed service. A consumer can then use the service's capabilities by invoking them.

Service loose coupling emphasizes on reducing the dependency between service providers and consumers. Loose coupling can influence the content and granularity of the

contract, which must also be implemented in a technology independent manner to enable interactions between providers and consumers built with different technologies. Loose coupling also influences the late binding of consumers to providers. Since a consumer is not bound to a specific provider, an invocation link is only created at execution time.

Service reusability stresses that services be considered as resources, where service capabilities are defined unconstrained to the functional context. For example, bundling an already existing application into a service enables this application to be reused in a service-oriented system. Non-proprietary technology enables service reuse.

A service is a deployable entity, independently of other services. It is also supposed to be autonomous and have a degree of control over the resources it uses, in order to enhance its reliability.

A stateful service involves the management of state information during consecutive interactions with a consumer (conversational state of many invocations). A stateless service does not depend on state information. Stateless service invocation does not relate to a specific client or to previous invocations. Service orientation recommends that services be stateless and become stateful only when required.

Complex services are built using the composition of many individual services. A composed service has a contract that uses capabilities from other services. Therefore, services must interoperate with each other, like consumers can interoperate with services. They should be able to effectively participate as members of the composition.

Service-oriented computing offers an abstract model to build distributed applications. It specifies a service as an abstraction and not simply as an interface like in traditional distributed computing. A better abstraction provides better agility and reliability when providers offer their capabilities to consuming clients.

2.3.3 Fundamental steps of the service provision process

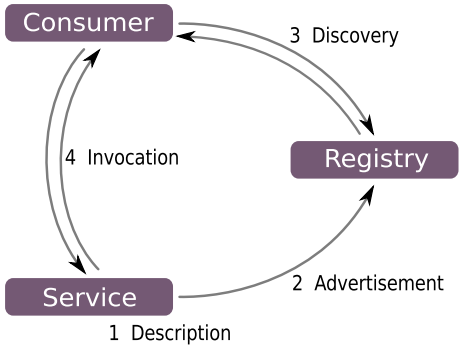


Figure 2.1: Entity interactions in a service oriented architecture

A service-oriented system follows the principles of the above-mentioned service-oriented design paradigm. A service-oriented system is built using an architecture called SOA (Service-Oriented Architecture). Entities in a service-oriented architecture are of three types: a **service** provided by a provider, a **registry**[1] helping services get discov-

[1]The word "registry" can be used interchangeably with the word "directory".

erable, and a **consumer** wanting to use the services (see Figure 2.1). The basic actions taken by these entities and the interactions between them —in order to create a viable service-oriented system— are:

- Description: The functional capabilities and non-functional properties of the service are described in a syntax understandable by all the entities of the system. It is also human-readable to enable easy creation. The description has well defined semantics to enable proper discovery. The description is created by the service programmer.
- Advertisement: The description is meant to be advertised by the service provider on a registry entity. The advertisement may add more information to the service description (e.g. provider availability times). This information is supposed to assist the consumer on whether or not he would like to use the service. The implementation of a service registry varies depending on the architecture of the system. Wired and stable networks use centralized registries where a provider registers its description, and where a consumer searches for registered providers. In distributed architectures, the registry can also be distributed using total or partial duplication over network nodes. In highly dynamic distributed environments, the registry can completely disappear in favor of a peer-to-peer type of discovery.
- Discovery: Discovery is a core principle of the service-oriented design. The consumer prepares a description of his needs, put together in a format resembling that of the service description. With these needs, the consumer seeks matching services. Discovery enables the matching between the consumer's needs and services of the registry having similar descriptions. The matching can result in many appropriate services, the consumer finally selects one service out of all these matching services.
- Invocation: Invocation enables a consumer to use a service. Once the consumer discovers an appropriate service, invocation consists of transmitting requests from the consumer to the service and then receiving responses back from the service. The communication link should be available during the interaction. Invocation requests and responses should be implementation-neutral in order to comply with the principles of the service-oriented paradigm. In other words, consumers and services built using different technologies should still be able to interact.

Section 2.3.4 enumerates the main interaction types and techniques that can be used to support invocation in general, and argues to what extent these techniques are appropriate in a service-oriented system.

2.3.4 Client/provider interactions in distributed computing

In the following, I enumerate the main techniques that are used in distributed systems to enable interactions between clients and providers [2]. I focus on the ability of these techniques to provide asynchronous interactions and less ties between the communicating entities.

[2]The word "client" can be used interchangeably with the word "consumer". The same can be done between the words "provider" and "service".

2.3.4.1 Message passing

Message passing is considered as the oldest form of interaction between distributed agents. An agent communicates by sending and receiving messages. It is a low-level interaction mode.

Message passing systems can be synchronous, where the sender and receiver have to wait for each other to transfer the message. In other words, the sender is blocked until the receiver has received the message, and the receiver is also blocked until the reception is over. Synchronous communications has the advantage of simplifying communications between senders and receivers. Message passing systems can also be asynchronous. To deliver a message from sender to receiver, the sender is not blocked, but still the receiver is blocked since it is listening synchronously on the communication channel. This communication channel is setup before any message exchange, therefore the receiver must be known to the sender and both of them must be active at the same time.

2.3.4.2 Remote procedure call

Remote Procedure Call RPC [29] was first introduced in procedural languages. Since then, it has been implemented in object-oriented systems as remote method invocations (e.g. Java RMI [30], DCOM [31], and CORBA [34]). The invoking object holds references to the invoked object, and remote interactions appear as local interactions. The client calls a client stub as a local method call, the stub puts the call parameters into a message, the message is sent over the network to the server, the server stub calls the server method, and the server reply is sent back to the client using a similar behavior. The SOAP [38] invocation specification is considered as message-passing, since it is a protocol for exchanging structured information in Web Services [39], but the message contents are usually RPC between the sender and the receiver.

Usually in RPC, the client sends a synchronous call (and is blocked until the reception of a response), the server can process the call asynchronously. To remove the synchronism from these remote invocations, some systems allow the caller to send one way calls and not expect a response, other systems (e.g. Future interface of Java Concurrent [40]) allow the caller process to keep working and access the reply when it becomes available (using a call handle). In RPC, the two interacting entities must be active at the same time, and the caller holds remote references to the invoked entity.

2.3.4.3 Event notifications

Notification systems use events for client-provider interactions. A normal RPC invocation is broken into two one-way invocations, the client calls the provider and puts a callback reference to itself, the provider responds by calling the client back using the reference. The observer pattern [41] uses event notifications, where an object notifies its observers by calling their notify methods.

Event notifications remove the synchronism that blocks the invoking client in RPC, therefore the client provider interaction becomes completely asynchronous. But still, an event notification does not work unless both interacting entities are active at the same

time, and they always hold references to each other.

2.3.4.4 Tuple spaces

In distributed tuple space systems, the tuple space is either a shared memory or a data container, where a network node inserts tuples for other nodes to access and read concurrently. The Linda [42] coordination language introduced this notion of tuple space. A tuple space is a collection of ordered tuples that any network node can manipulate using three main operations: *out* puts a tuple in the tuple space, *read* reads a tuple from the tuple space, and *in* reads and removes the tuple from the tuple space. Lime [43] is an implementation of Linda for mobile environments.

In tuple spaces, the interacting entities do not hold references to each other in order to communicate. The node that inserts a tuple in the tuple space does not know who is accessing this tuple. And a node reading a tuple does not know who put it in the space. Furthermore, the read operation lets many nodes read a tuple, which enables a one-to-many communication mode. In addition, the interacting entities do not have to be active at the same time for the communication to succeed. But the synchronization problem remains, since a node is blocked when reading a tuple from the tuple space. Therefore, some implementations of tuple spaces like JavaSpaces [44] provide asynchronous notifications of new tuples.

Another close example to shared data is the flow-based programming [45] where applications are considered as "black boxes" that exchange messages through predefined connections. This flow-based programming is mainly used in component-oriented systems. Like in tuple spaces, interacting entities do not hold references to each other.

2.3.4.5 Message queues

Message-oriented middleware (MOM) [46] emerged from the need to glue together remote applications without re-engineering individual modules. A MOM depends on message queuing (e.g. Apache Qpid [47], JMS [48]) for interactions between clients and providers. A queue is a storage space where messages can be placed by producers, and then concurrently pulled by consumers. The queue stores messages in a first in first out (FIFO) manner, and provides transactional and ordering guarantees. To some extent, message queuing systems relate to the publish/subscribe interaction model.

Message producers and consumers do not share references to each other, and they do not have to be active at the same time in order to communicate. A producer is not blocked when storing messages, but a consumer is blocked when pulling these messages.

2.3.4.6 Publish/Subscribe

A publish/subscribe system [24] lets producers send messages to a message broker via a *publish* method, the broker also has a *subscribe* method to let consumers subscribe to messages of interest. The broker filters published messages according to the interest of each of the subscribers, and notifies these subscribers of matching messages. There are two main techniques of filtering: topic-based and content-based. In topic-based pub/sub

systems, a topic is a keyword representing a class of messages, each published message belongs to a topic, subscribers subscribe for the kind of topics they are interested in. When a producer publishes a message, the broker notifies the subscribers of the message's topic. The broker is often centralized, it may also be distributed, it may sometimes disappear in favor of a peer-to-peer interaction model. One of the oldest topic-based pub/sub systems was Isis [49], a more recent example is TIBCO Rendezvous [50]. In content-based pub/sub systems, the subscription is based on the actual contents of a published message. The broker filters messages according to their internal attributes or their meta-data (e.g. Siena [51], Jedi [52], JMS [48]). Another example is CORBA with DDS [53] where the client-server model in CORBA is extended using the publish/subscribe features of a DDS data distribution service.

With pub/sub systems, the interacting entities do not need to know or hold references to each other. A publisher does not know who and how many are receiving its message, and the subscriber does not have to know who published the message. Furthermore, a communication succeeds even if the entities are not active at the same time. In addition, the publisher is not blocked when publishing a message, and a subscriber is not blocked when subscribing and gets notified at message reception, therefore the communication is completely asynchronous.

2.4 Service provision systems

Computer devices might be using an infrastructure to be connected to the Internet or to local networks, or else they might be using ad hoc communications without infrastructures. Taking into account the characteristics of network environments, service provision systems can be divided into three types: those designed for stable networks, others designed for dynamic networks, and others designed for MANETs.

2.4.1 Service provision systems for stable networks

2.4.1.1 OSGi

The OSGi [54] platform is maintained by the OSGi Alliance. Its is not intended to create distributed applications, it is a middleware that defines a dynamic component system for Java, where local applications are dynamically composed of many reusable service components. A component is created by developers and put in a repository as a software bundle ready for deployment. A bundle is a collection of Java classes with a manifest description containing its identifiers and dependencies. A bundle is deployed as a service on a local OSGi capable machine, and its service interface is registered at the local service registry. The life cycle of a bundle is controlled using a simple API (install, start, stop, update, uninstall). Local services can discover other local services from the local registry, in order to fulfill their dependencies. The OSGi middleware is intended to enable a local service-oriented platform. Though, bundles can exist on remote software repositories ready to be discovered and downloaded. In addition, a bundle's life cycle can be remotely controlled. Furthermore, the R-OSGi [55] proposition enables interactions between remote services deployed on different machines. OSGi is used as the base

platform for many Java EE application servers (e.g. GlassFish [56], JBoss AS [57], JOnAS [58]).

2.4.1.2 Enterprise services

Since the 90's, enterprise service architectures have helped shape the principles of the service-oriented paradigm. These architectures range from application integration, to process management, to complex service-oriented architectures.

EAI Enterprise application integration is a middleware framework composed of a collection of technologies to enable integration of systems and applications across the enterprise. It abstracts proprietary applications through the use of adapters, brokers, and orchestration mechanisms. An example is the enterprise service bus ESB [59], where the bus is a messaging engine that enables applications to communicate using technologies like XML and SOAP (e.g. OpenESB [60], WebSphere ESB [61], etc.).

BPM Business process management considers assets of the system as processes. It focuses on on the management, automation, and adaptation of these processes. It assumes the role of the process composition controller. It is implemented using orchestration technologies like BPEL [62].

SOA Enterprise services matured with the creation of complete SOA platforms by various vendors or open source projects (e.g. Oracle Fusion Middleware [63], GlassFish [56], JAVA EE [35], SAP SOA [64], etc.). The platforms offer varying functionalities like discovery, integration, process management, messaging. They use a wide range of technologies like RPC, SOAP [38], REST [65], DCOM [31], CORBA [34], SCA [66], and Web services [39].

2.4.1.3 Web Services

The service-oriented paradigm is an abstract and implementation-neutral paradigm. Still, its association with Web services [39] has become very common. A reason for this association, is that the majority of SOA vendors have modeled their service-oriented platforms as Web services.

By definition [3], a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. A Web service has an interface described in the WSDL machine-processable language. Other systems read the WSDL description in order to interact with the Web service using SOAP messages. These messages are usually transported over HTTP using XML serialization.

The owner of a Web service (a person or organization) is called a *provider entity*, it provides the functionalities of the service through the implementation of a *provider agent*. Likewise, a *requester entity* is a person or organization wishing to use the service, the *requester agent* software exchanges invocation messages with the *provider agent* software.

[3] http://www.w3.org/TR/ws-arch/#whatis

But before invocations, the request and provider entities become known to each other, and they agree on the service description and semantics that determine the interactions between the requester and provider agents. This is the discovery phase that can be implemented using three approaches: (1) using a registry service that is centrally controlled by an authority where providers actively register their services, (2) using the index approach where an index is automatically maintained by a Web crawler for example, (3) using peer-to-peer discovery where a requester directly queries the providers to search for services.

Web Service Description Language WSDL [67] was originally developed by IBM, Microsoft, and Ariba, the current version WSDL 2.0 is a W3C recommendation. A WSDL document describes a Web service using the XML format. It contains an abstract section and a binding section. The abstract section describes the interface composed of operations, each operation defines its input and output messages. The abstract section also contains the data types of these inputs and outputs written in XML Schema. The binding section binds the operations and messages to a concrete network invocation protocol (e.g. SOAP binding, HTTP binding, etc.). In order to enhance the discovery, the WSDL document can include semantic annotations in SAWSDL [68]. Semantics help categorizing the interface when publishing to a registry, and therefore it helps requesters with discovery. Semantic annotations can also map data types from an ontology during invocation.

Simple Object Access Protocol SOAP [38] was originally developed by Microsoft, the current version SOAP 1.2 is a W3C recommendation. It is a protocol specification for exchanging invocation messages between requesters and providers. The message format is XML. SOAP relies on RPC for the invocation model (request and response messages). Messages are usually transmitted over HTTP, but there is other transmission modes like JMS [69] and SMTP. A SOAP message is an XML envelope containing an optional header part and a body part. SOAP helps create stateless or stateful Web services, where a service may expose an arbitrary set of operations. There exists another type of Web services called REST-ful Web services. REST [65] enables service invocation but only using a uniform set of stateless operations over HTTP (create, retrieve, update, delete).

Universal Description Discovery and Integration UDDI [70] was originally developed by IBM, Ariba, and Microsoft. The current version 3 is sponsored by the OASIS organization. UDDI specifies the creation of a central registry called UBR (UDDI Business Registry), and specifies a framework to describe service metadata and another framework to discover them. An XML document describes the capabilities and properties of a service. Description can include information about the business providing the service like its name and contact information of administrators, it also includes identifiers for the service and descriptions. Registration at the UBR is made using SOAP over HTTP, and service requesters query the UBR for services. UDDI is mostly implemented for the enterprise world.

In comparison to traditional distributed systems, Web services are appropriate for applications operating over the Internet where reliability cannot be guaranteed. In addition, components of the system run on different technologies, platforms, and vendor

products. Web services help wrap and expose existing applications over the network. Web services do not require to synchronously manage deployment at all consumers and providers. And finally, Web services can be composed and orchestrated (WS-CDL [71], BPEL for Web services [62]).

2.4.1.4 Service Component Architecture

The Service Component Architecture (SCA) [66] specifications describe a model for building applications using a service-oriented architecture. It represents an extension to service implementation approaches like Web services. The model consists of composing components, where each component offers its services and requires references to other services. SCA aims to provide a wide range of implementation technologies for component creation, like different programming languages and frameworks. SCA also aims to provide a wide range for the access methods connecting these components, including different communication technologies like Web services, messaging systems, and RPC.

2.4.2 Service provision systems in dynamic networks

Dynamic networks are characterized by network nodes that can join or leave the network due to their mobility. Typical examples are local networks in home and enterprise environments, where computers can be connected using wired LAN or wireless WiFi hotspots, or using Bluetooth ad hoc communications. In these environments, connections and disconnections happen at a slow rate. Therefore, service provision systems focus on discovering newly connected nodes and the services they offer. Once discovered, invoking these services is not considered as a challenge because a network connection is available between the service and the consumer.

Jini [72] was originally developed by Sun Microsystems, it is now maintained by the Apache Software Foundation under the Apache River project. It is a service-oriented architecture for building secure distributed systems. It defines a programming model that extends the Java technology. The Jini architecture consists of three types of entities: service providers, clients, and lookup services. A service can be a software component, a hardware device, or a combination of the two. A service provider uses multicast to find a lookup service, and then registers a proxy object of its service. A client also uses multicast to find a lookup service, then it specifies an interface that the desired service might implement. If a matching service is found, the client copies the proxy object of the service. This proxy object is then used to directly invoke the service provider. Jini uses serialization to send objects over the network. Services can be grouped into federations to allow clients to search for specific groups. Multiple lookup services can coexist to prevent failure points.

Service Location Protocol SLP [73] was developed by the SRVLOC work group of the IETF, as a service-oriented computing standard. A service agent describes the location (URL) and capabilities of a service (attribute-value pairs), and registers at a directory agent. A user agent queries the directory agent using attributes in order to discover

matching services. Services can be grouped into scopes, and clients can browse all available services. If there is no directory agent, client and service agents can still perform peer discovery using multicast. When a directory agent becomes available, it multicasts its presence for the client and service agents to use. The SLP standard does not specify an invocation protocol.

DNS Service Discovery DNS-SD [74] is an IETF Internet draft. It extends the DNS [75] Domain Name Service protocol by adding service attributes to DNS records to allow clients to receive service information through normal DNS communication. DNS-SD is used in the Zeroconf stack.

Universal Plug and Play UPnP [76] was developed by Microsoft and is now maintained by the UPnP forum. When a device joins the network, it advertises its presence, so it could present its description of capabilities on demand (XML descriptions). A control point acts as a directory that collects device descriptions. The control point discovers devices using SSDP [77], then it asks for a detailed description of the device's services, then it uses SOAP to discover and invoke specific actions. If the state of a service changes, the GENA [78] notification architecture reports events to the control point. UPnP is deployed over TCP.

Salutation [79] is a discovery and invocation model from the Salutation Consortium. It enables devices to advertise their capabilities expressed as a collection of attributes. A device has a local SLM manager for storing its service descriptions. The SLM managers on different devices communicate to exchange service descriptions. Communications between SLMs are independent of the transport protocol, they can work over many transport managers TM, each TM implementing a transport protocol. Periodic checks ensures that a client has a list of the currently available services. Invocation is supported using RPC.

Bluetooth Service Discovery Protocol Bluetooth SDP [80] is a service discovery protocol for Bluetooth devices in proximity (only one hop communications). Service characteristics are described using attribute-value pairs. A client device has the option of searching for specific services by querying a provider device using desired service attributes, for the provider to match and return relevant services. Otherwise, using a generic query, the client can ask the provider for a list of all available services. The protocol does not specify advertising, service directories, or service invocations.

In addition to the presented systems, there exists many others that are used for dynamic networks. For example IGRS [81] which is comparable to UPnP, uses an extended version of the SSDP service discovery. Another example is EchoNet [82] which is intended for device interoperability in a home or a building setting.

2.4.3 Service provision systems in MANETs

The design of service provision systems for MANETs is inspired by some of the previously described systems for stable and dynamic networks, yet it takes into account the special characteristics of mobile environments. I enumerate some selected provision systems for MANETs, and then I discuss their various characteristics.

DeapSpace [83] is a service discovery protocol for single-hop ad hoc networks. A service is described using a compact predefined format containing its name, address, inputs, outputs, properties, and time to live. Each node maintains a list of its local and discovered service descriptions. A node advertises its list of service descriptions to its neighbors using periodic broadcasts. A node receives broadcasts and adds the advertised service descriptions to its service list. Using these broadcasts, a service node can have a view of all the services available in the network.

Konark [84] is a discovery and invocation protocol for multi-hop ad hoc networks. A service description is based on WSDL. Each node maintains a local service registry that stores service descriptions in tree based structure. The protocol uses multicast to advertise and discover services. When a client node needs a service, it multicasts a search query. If another node receives the search query and finds matching services in its registry, it advertises a message for each service. Service advertisements use multicast or periodic broadcasts, if some other node picks up an advertisement, it adds the service description to its registry. A provider node has a micro-HTTP server that handles client invocations. Invocation requests and responses are based on SOAP over HTTP.

Chakraborty et al. proposed the GSD [85] service discovery protocol for pervasive and ad hoc environments. A service is semantically described using the OWL [86] ontology language; it enables the hierarchical grouping of services. A provider node broadcasts service advertisements to the vicinity using one hop broadcasts, and nodes receiving the advertisement can relay it according to a specified number of hops. Service advertisements are also cached in the receiving nodes. A node can also periodically advertise information about the service groups found in its vicinity, so that service group information is propagated to the whole network. A client node can also initiate a service discovery request that can be selectively forwarded from node to node depending on the service group information, until it reaches a provider node. A discovery response is routed back to the client on the same request route, or using a routing protocol in case of failure. Chakraborty et al. also proposed a service invocation protocol GSR-S [87], that uses the discovery routes created by GSD to support invocations. It specifically uses the route created by advertisements or the route created by the discovery request, and assumes that the links are reversible. It also enables session maintenance, where a client can initiate a session with a specific provider node or with the service instance independently of the provider node.

Sailhan et al. proposed a scalable service discovery protocol for highly dynamic large-scale ad hoc networks [88]. Some nodes of the network may have multiple wireless in-

terfaces, enabling them to be connected to the Internet. A service is described using WSDL enriched with QoS parameters. QoS parameters are service-related (e.g. security, availability time, reliability), and resource-related (e.g. performance, memory, energy, geographical location). A subset of the network nodes act as service directories, and form a virtual network. A directory node caches the description of the services available in its vicinity, and it advertises its presence to its neighbors. Directory nodes communicate with each other. Each directory periodically broadcasts its profile summarizing the cached services to peer directories. When a client node needs a service, it queries its closest directory node. If the directory node does not have a matching description, it searches in the received profiles of its peers, and contacts peer directories that are likely to have a matching description. If found, the description is transmitted to the origin directory and then to the client. Directory nodes are uniformly and dynamically deployed within the network.

PDP [89] is a pervasive discovery protocol for ad hoc networks with limited device resources. A node contains a service cache containing a list of known services. A service is described using a description language called GSDL based on XML, it enables the description of hierarchical relationships among services. A service provider node broadcasts the descriptions of its services not periodically but only when a client asks for them. All devices in the broadcast range store them in their cache. When a provider device receives a discovery request from a client device, it checks if it is one of the provided services or listed in its cache. If found, the provider waits for a calculated amount of time before sending its discovery reply. During this time, the provider listens to other providers advertising the same service type. A client node can send a discovery request to one device only or to all devices providing the same service type. When a provider node wants to switch off, it sends a deregister message to the other devices. Invocations are implemented using SOAP requests and responses.

LSD [90] The Lightweight Service Discovery protocol is implemented over the OLSR proactive routing protocol. A client's discovery request and response are piggybacked in the messages of OLSR, as well as the advertisement and registration messages. LSD supports the presence of a service directory, it periodically advertises its services. A client searches for a directory, if it does not find one, it queries providers directly. Service invocation is considered as stable, the SIP protocol was used in simulations.

Handorean et al. [91] uses context-aware information to support predictable service provision in MANETs. Discovery is implemented using tuples spaces. A tuple is a sequence of Java objects (each has a type and a value). A node places a tuple in the tuple space to make it available to all other nodes that are sharing the same tuple space. Tuple spaces having the same name are merged when their nodes are within one hop communication range. A node needing to read something from the tuple space creates a template describing the interesting tuple. The template should match a tuple in order for the node to read the matching tuple. The tuple space system enables a coordination model. It enables a client node to discover the available provider nodes in its proximity. The system implements a session management protocol called "follow-me". A client starts by invoking a neighboring provider by opening a session. When the client moves, the session

is migrated to another close by provider that can be accessed by the client. In case of a disconnection, the session can be reopened at another provider when one come into the client neighborhood. The system uses strong process migration, everything needed for the execution of code on another node. Sometimes, volunteer nodes can play the role of a service provider. The system only uses one hop communications.

2.4.4 Discussion

There exists a broad diversity of provision systems for MANETs, the previously described ones are a selected representation of this diversity. Many surveys and comparative studies are proposed in the literature [92, 93, 94, 95] that try to classify these systems according to various criteria. In the following, I present the prominent design characteristics of these systems.

Emphasis on discovery.

Provision systems are primarily intended to support network-wide service discovery. The goal is to enable every client node to discover service providers with a high success rate. In addition, systems try to minimize the network traffic induced by the discovery protocol using different combinations of broadcast, multicast, or unicast transmissions. The duration of the discovery is also taken into account, that is, the time between the moment when the client issues its discovery request and the moment when this client receives a discovery reply. This duration depends on the topology and the density of the network. It can be of the order of seconds in some connected environments.

Design level.

Some provision systems are designed so that the service discovery is directly dependent on an underlying routing protocol. These protocols are referred to as network layer protocols or cross-layer protocols. Routing protocols transmit node discovery packets in order to create routing tables, discovery can just use this already available information to enable communications between service clients and providers. Therefore discovery requests and replies are piggybacked to the messages of the routing protocol. Once a client node discovers a provider, end-to-end invocation interactions are also carried out using the routing protocol. Some of these systems work over known routing protocols (e.g. LSD, AODV-SD [96], and others [97, 98]). Other systems build the routes of communications and service discovery at the same time (e.g. GSD [87], and [99]). Network layer protocols usually make use of limited service descriptions, but significantly reduce the network overhead due to their use of the routing information.

Other provision systems are designed independently of the the underlying communication protocol. They are called application-layer protocols. The discovery protocol involves using some form of vicinity advertisements (broadcasts, multicasts), but mostly it still needs an underlying routing protocol to enable point-to-point communications (e.g. DeapSpace, Konark, Sailhan et al., PDP). Yet these systems stay independent from the routing layer. Application-layer protocols offer better portability than the network-layer protocols.

Service description.

The minimal method to describe a service is using unique identifiers UUID (Universal Unique Identifier) that should be known to all network nodes. This minimal description is mostly used in network layer discovery systems. More attributes can be added to the UUID to enhance the description. A small size description helps lower energy consumption and network bandwidth. Meanwhile, other systems enable enriched service descriptions in order to help clients with service selection. These descriptions mostly use attribute-value pairs written in XML-based languages, describing functional (e.g. Konark) and appending non-functional service properties (e.g. Sailhan et al.). Non-functional properties are mostly QoS properties to help the client select the closest provider node or the most available one (e.g. [99]), in an effort to limit long-distance and failure-prone invocations. Descriptions can also help classify services according to predefined trees (e.g. Konark, PDP). Other descriptions use semantic ontologies (e.g. GSD), and categorize services into groups. Service selection is usually done manually by the client, except for some systems where selection is automatically incorporated into the discovery protocol, like GSD that uses the group information to properly forward discovery queries, and [100] that effectively use geographic proximity information.

Directory architecture.

Among application-layer discovery protocols, the service directory is designed using various architectures. Centralized directory architectures are excluded, because a directory hosted by a single node cannot always be accessible to all nodes of the network, it also introduces a single point of failure, and does not necessarily have enough resources to serve all network nodes. Therefore, all discovery protocols use distributed directory architectures with two main variations: peer-to-peer, or using overlay networks. In peer-to-peer only architectures, each node has a cache containing a list of its services and descriptions of others found in the network. The main disadvantage is that the service descriptions must be periodically advertised from each node (broadcasts or multicasts). Descriptions travel from node to node in order to properly disseminate to all network nodes, which produces large amounts of network traffic. Examples of such systems are Konark, GSD, and others [101, 102]. In systems using overlay networks, the directory is distributed on only some nodes that form the virtual overlay network. Distribution can be done using fully duplicated directories or partially duplicated but coherent directories. The overlay nodes are dynamically an uniformly deployed in the network, so that any client can have access to a nearby directory node. Directory nodes collect service descriptions and communicate with each other to maintain the known service list, which restricts heavy discovery communications to the overlay network only. Therefore these systems are scalable and can handle high node density, but on the downside it is difficult to maintain directory coherence in case of highly dynamic networks. Examples of such systems are Sailhan et al., and others [103, 104, 105, 106].

Discovery interaction modes.

There exists two main modes of discovery interactions between clients and providers: push or pull modes. In push mode also called reactive discovery (e.g. DeapSpace), providers push their service descriptions using unsolicited advertisements, and clients listen and select the services they need. Provider advertisements are usually broadcast to

the one-hop neighborhood, or multicast with multi-hop support. In pull mode or proactive discovery, a client asks for needed services by sending a discovery request directly to provider nodes or to directory nodes. The discovery request is either sent using broadcast, multicast, or unicast. A provider or directory answers back with matching service descriptions. But the majority of discovery systems use both push and pull modes (e.g. Konark, PDP, GSD).

Invocation support.
The above-mentioned provision systems offer varying support to service invocation. After discovery, the discovery reply obtained by a client is an address of the provider, and subsequent invocations are not considered as a challenge. A large majority of systems designed for connected MANETs rely on dynamic routing protocols, using one-to-one or one-to-many invocations. But few provision systems present characteristics viable in disconnected MANETs, that is without routing or overlay directory networks, but still they do not enable network-wide invocations. Some only enable proximity invocations (e.g. UPnP), where a client and a provider are neighbors. And others enable proximity-only invocations with service migrations to maintain service sessions (e.g. [91]). Other solutions propose to handle service disruptions using dynamic service-oriented computing. In dynamic SOC, services register and unregister themselves at the directory according to their contextual availability, and clients therefore can dynamically use the available provider. [107] proposes service level agreements (SLA) for dynamic SOC, where the client and the provider agree on service disruption concerns.

2.5 Conclusion

In order to design a middleware platform that supports the execution of service-oriented applications for mobile nodes in disconnected MANETs, I studied the service-oriented computing approach and how communications can be provided in the targeted environments. This chapter presented the state of the art of both communications and service-oriented systems in mobile ad hoc networks.

MANETs are spontaneously formed networks that do not need any form of infrastructure. In connected MANETs, communications are implemented using dynamic routing protocols, where routing tables are created and maintained using either proactive or reactive routing techniques. In disconnected MANETs, the network contains disconnected islands where routing protocols do not work. In these disconnected networks, communications are tolerant to disruptions, and protocols mostly use a store-carry-and-forward mechanism to enable network wide communications. Protocols rely on data dissemination from network node to another. Protocols can support destination-based or content-based communication styles.

A distributed system consists of multiple autonomous computers that communicate through a computer network in order to achieve a common goal. After object-oriented and component-oriented distributed systems, service-oriented systems provide greater benefits for the loosely coupled mobile computing. Service-oriented computing systems follow the principles of the service-oriented design paradigm. The fundamental elements needed to create a service-oriented system are: service description (providers describe

their provided services), advertisement (these providers advertise their services), discovery (clients discover advertised services), invocation (clients use the provided services). A client uses a service using invocation interactions with a provider. Few of the techniques that are used to enable invocations in distributed systems in general are adapted to the needed loose coupling of a service-oriented system. Service provision systems vary according to the target network environment, the majority are designed for stable networks (mainly enterprise and web services) and more dynamic networks (like local networks). Even though there exists service-oriented systems specifically designed for MANETs, issues related to service discovery, advertisement, invocation, still have their own specific challenges. The description of my solutions to these challenges follows in the next chapters.

3

Challenges and Design Overview

Contents

3.1 Introduction 29
3.2 Challenging points 30
3.2.1 Communication 30
3.2.2 Interoperability 30
3.2.3 Service contract 31
3.3 Design overview 31
3.4 Summary 32

3.1 Introduction

To cope with continuously changing heterogeneous environments, Service-Oriented Computing (SOC) emphasizes the construction of an abstract model for distributed systems. In comparison with traditional distributed systems, abstraction provides better agility and reliability when clients consume the capabilities offered by providers. In other words, SOC specifies a service as an abstraction, not simply an interface.

The main two features of my service provision system must be (1) the ability of service entities to tolerate long periods of disconnection, and (2) the ability to handle change. This can be achieved by making the different entities of the architecture less dependent by loosening their strong ties with the other entities. Being less dependent means that each service entity is more autonomous, which in turn means less needed interactions to complete a service task. This is essential in a disconnected mobile environment where an interaction between two nodes can take a considerable time to finish, due to the lack of deterministic and instantaneous communications. Additionally, loosening the ties means more tolerance to change and failure. The ability to handle change can help improve the performance of the service provision. In order to design a viable service provision system, loose coupling is highly desirable on many challenging points. The following sections enumerate these points and describe the design overview.

The notion of *"Loose Coupling"* was first introduced by Karl Weick [108] in the field of organizational studies. According to Weick, coupling represents the degree of direct knowledge that an organization has of another. By loosening this coupling, one organization becomes less dependent on others.

This notion of loose coupling is currently used as a principle of the service-oriented paradigm. It has effectively influenced the late-binding of consumers to providers. It has also influenced the use of technology-independent interaction schemes between consumers and providers. Throughout the rest of this document, I intend to improve the loose coupling between service entities. The looser the coupling that the service platform's design can achieve, the better it can handle the challenges of disconnected environments.

3.2 Challenging points

3.2.1 Communication

In order for the service provision system to tolerate disruptions, a communication layer has to provide mechanisms to decouple two service entities in terms of temporaneous interaction, synchronous behavior, and mutual knowledge. The following time-synchronization-space three dimensions of communication decoupling was identified in [24].

- Time: Since mobile nodes exhibit unpredictable volatility in their environment, there is a high probability that two service entities are not active at the same time. In addition, two entities residing in different network islands cannot have a connected communication link. Hence the communication problem in case a consumer requests information from a non-active or unreachable producer, or conversely in case a producer sends information to a non-active or unreachable consumer. Therefore, a temporaneous interaction between two entities is not guaranteed.
- Synchronization: Because of the temporaneous interaction problem, disconnected mobile communications are inherently asynchronous, therefore a service provision system must naturally handle this. A producer must not be blocked after producing its information, and likewise a consumer must not be blocked while it is waiting to receive this information.
- Addressing space: Decoupling the addressing space means that two interacting entities do not need to know and hold references to each other. Even more, a producer of information does not know how many consumers are consuming this information. Likewise, a consumer of information does not know how many producers it is getting its information from. A publish/subscribe communication system is capable of applying this space decoupling between service entities that thereupon communicate as decoupled publishers of and subscribers to information.

3.2.2 Interoperability

Service providers and clients should always be able to understand each other regardless of their implementation technology. The abstraction of the implementation can only be achieved using universal standards and specifications for the sake of interoperability. A service provider implemented in Java should seamlessly interact with clients implemented in C++ or .NET for example.

This implementation abstraction is well done in service provision systems like Web services [39] using XML as an interaction language, and specifically using specifications like WSDL [67] and SOAP [38] for service description and invocation. The service platform proposed in the following chapters also uses WSDL and SOAP in the discovery and invocation interactions.

3.2.3 Service contract

By hiding the implementation technology, a change in the implementation of a service provider does not mandate any change at the consumer side. But if the service contract changes (like a change in function behavior), this change of course impacts the consuming clients. The service provision system has to manage these changes to decrease the loss of service from a client's point of view. Therefore the service-oriented paradigm enforces late-binding between consumers and providers in order to free their coupling until the last moment before invocation. Yet during the invocation, the service contract binds the consumer to the provider. In other words, the consumer sends its invocation request to the specified provider destination. This type of request can cause the failure of the invocation in case the specified provider is unreachable. A provider can become unreachable to the client because of the very nature of disconnected MANETs.

The service platform proposed in the following chapters presents content-based invocations. A content-based request is created according to a discovered provider, but it can however be answered by other providers offering compatible service contracts.

3.3 Design overview

The global objective is to build a service middleware platform for mobile nodes that supports the execution of service-oriented applications in disconnected MANETs. In order to separate the concerns proper to communications and those proper to service interactions, the service platform's design consists of two layers: a communication layer and a service layer. Figure 3.1 shows that a mobile network node can for example consist of a human user carrying a mobile computer where a service platform is installed.

The service layer called DiSWAN (Distributed Services in Wireless Ad Hoc Networks) contains all the service agents. A service agent is either a client or a provider. The platform can host many clients and many providers. A service agent communicates with other agents on remote nodes via the communication layer. A service agent acts on behalf of a higher-level user application. A user application may perform automatic actions or require the intervention of a human user. The definition of the application level is out of the scope of this work, and is simply referred to as "user" in the remaining of the document.

The following chapters describe the architecture of the service platform. The architecture can be viewed through two different design patterns. Each pattern focuses on a different aspect of interoperability between service entities.

- The **Message Oriented** pattern focuses on the interactions of service entities using a message oriented communication model. The format and structure of messages

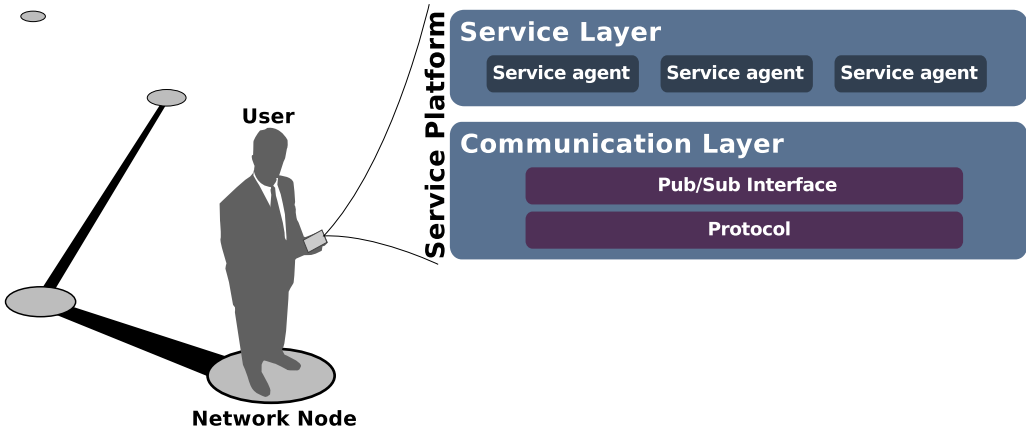


Figure 3.1: Two layer service platform on each mobile network node

in terms of headers and bodies is defined to enable service entities to understand the exchanged messages. In addition, this pattern defines how these messages get transported by network nodes as determined by a communication protocol. Chapter 4 describes the communication layer used in order to take care of the communication challenges of time-synchronization-space. The communication protocol ensures a network-wide content-driven dissemination of messages. The protocol implements disruption-tolerant information caching enforced by a content-driven and opportunistic gossiping system. The communication layer provides a publish/subscribe interface accessible to higher-level service agents.

- The **Service Oriented** pattern builds on the basics of the message oriented communication pattern by adding the concept of action and service. It interprets messages as requests for actions and as responses to those requests. The model therefore allows the messages to be expressed in order to yield different expectations (e.g. request message, response message, service descriptor message). The discovery and invocation protocols described in Chapters 5 and 6 define the service-level interactions of the platform's service layer. These protocols draw inspirations from Web service technologies in order to enable interoperability between service entities. In addition, discovery relies on the information dissemination properties of the underlying communication layer. Furthermore, invocations use content-based requests in order to enable late-binding and addressing multiple providers.

3.4 Summary

I presented in this chapter the challenging points that must be addressed for the design of a service provision platform that remains viable in disconnected mobile environments. The design focus is to achieve better entity independence and loose coupling at each of the presented points, despite the challenges inherited by mobile environments in comparison to infrastructure and static environments. Consequently, the aim is to respect these principles throughout the decision making in the design process of the communication, discovery, and invocation protocols.

4

Disconnected Communication Support

Contents

4.1 Introduction 33
4.2 DoDWAN communication protocol 34
4.2.1 Opportunistic gossiping 34
4.2.2 Periodic announcements 35
4.2.3 Local cache 35
4.2.4 Content-based matching 35
4.2.5 Mobility as an advantage 35
4.2.6 k-hop opportunistic gossiping 36
4.2.7 Frugal use of the wireless medium 36
4.3 Publish/Subscribe interface 36
4.3.1 Message 36
4.3.2 Publishing a message 37
4.3.3 Subscribing to messages 38
4.3.4 Canceling a message 39
4.4 Discussion 39
4.4.1 Communication delay 39
4.4.2 Loose coupling benefits 40
4.5 Summary 40

4.1 Introduction

The service platform consists of two layers: the service layer containing service agents, and the communication layer. The needs of the service platform concerning the support for communications are:

1. Counterbalance the unpredictable availability that excludes temporaneous interactions between service providers and consumers.
2. Enable asynchronous and unblocking service requests and responses.
3. Decouple the references between providers and consumers.

For the platform's communication layer, I used the DoDWAN [25, 26] protocol for disconnected MANETs (Document Dissemination in mobile Wireless Ad Hoc Networks). DoDWAN was developed in my research team (CASA) of the Valoria laboratory at the "Université de Bretagne Sud" [1]. DoDWAN implements an opportunistic and content-driven communication protocol, accessible to higher layers through a publish/subscribe interface. A high-level service agent can therefore publish and subscribe to structured pieces of information called messages. In order to receive interesting messages, a service agent subscribes using a predicate to match the contents of these messages. The union of subscription predicates form a mobile node's interest profile. A network node has a local storage space (cache) to store messages. This storage space enables the store-carry-and-forward mechanisms. In other words, the communication protocol enables mobile nodes to store and transport interesting messages, so that a published message disseminates amongst mobile nodes according to wireless contact opportunities and to the interest profiles of these nodes.

Section 4.2 describes the communication layer's protocol, which uses caching, content-based matching, and opportunistic gossiping. Section 4.3 presents the communication layer's publish/subscribe interface, it describes what a message is and how a message can be published, subscribed to, and cancelled. Section 4.4 describes the resulting characteristics of the communication layer, stressing that it is a best effort solution that induces communication delays. Section 4.5 concludes the chapter.

4.2 DoDWAN communication protocol

DoDWAN is built as a middleware for mobile network nodes. The middleware implements a communication protocol accessible to the service layer through a publish/subscribe interface. The entities and principles of the protocol are detailed in the following (further details can be found in [26]).

4.2.1 Opportunistic gossiping

In a dynamic mobile environment, the mobility of nodes yields contact opportunities between them. The protocol implemented in DoDWAN exploits these transient contacts in order to opportunistically exchange messages. Each node in the network is associated a "profile", that determines the kind of information it is interested in. A gossip-like mechanism orchestrates interactions between neighboring nodes, allowing them to exchange messages according to their respective interest profiles.

Interaction between mobile nodes relies on a simple scheme, whereby each node periodically announces its profile and a "catalog" of the message headers that are currently available in its local cache. When a node discovers that one of its neighbors can provide a message it is interested in (that is, a message header that matches its own interest profile and that is not already available in its own cache), it can request a copy of this message from this neighbor. At the time of the reception of this request the neighbor broadcasts

[1] Specifically, DoDWAN represents the work accomplished by Julien Haillot for his thesis "*Définition et validation d'un modèle de communication supportant la communication basée contenus dans les réseaux mobiles ad hoc discontinus*".

the corresponding message, thus all nodes located in its neighborhood can benefit from this broadcast. Any node that receives a message verifies if the header of this message matches its own interest profile and, if so, it additionally verifies if this message is not already present in its local cache. If this is not the case, then it puts the newly received message in its cache. Upon receiving one or several requests for a particular message from its neighbors, the owner of this message broadcasts it on the wireless medium, so it can be received by all requesters simultaneously.

Transient contacts between mobile nodes are thus exploited opportunistically for exchanging messages between these nodes, based on their respective interest profiles, and based on the messages they can provide each other on demand.

4.2.2 Periodic announcements

The gossiping mechanism relies on announcing the interest profile and the message catalog of each node. If two nodes are passing by each other, the time window of a possible contact gets slimmer with a higher node moving speed. Therefore, announcements are repeated in a periodic way so that two nodes detect each other inside of the contact time window, and with additional spare time for any eventual message exchange.

4.2.3 Local cache

Each node in the network has a dedicated storage space serving as a local cache of messages. A message created by the node is deposited in this cache, and messages received from other nodes are also stored in the cache. Since this storage space is necessarily limited (especially on small mobile devices), the cache purges messages that have reached their end of life (deadline), and in case the cache grows full it purges older messages to make space for new ones.

4.2.4 Content-based matching

Message exchange between mobile nodes relies on content-based matching. A mobile node receives only messages of interest according to the contents of the message. When a message is created at a node A, it is handed over to other nodes that are interested in the content of this message. For example a node B that expressed its interests in its profile, will retrieve the message when it gets the opportunity if the profile matches the message. Likewise, the message is also transferred from the interested node B to other interested nodes that may not have direct contact with A.

4.2.5 Mobility as an advantage

By storing messages in the local cache, a node can serve as a mobile carrier for these messages while moving in the network ("store-carry-and-forward" principle). In a disconnected environment, a carrier node becomes the physical relay of information. The success of communicating a message from creator node A to an interested distant node C residing in a different network island depends on intermediary relay nodes.

4.2.6 k-hop opportunistic gossiping

In addition to the one-hop gossiping between two neighbor nodes, the protocol extends the gossiping phase to the nodes of a connected network island using k-hop forwarding. This is efficiently implemented using multi-point relay selection. Requests are sent using multi-point broadcast and replies are sent using unicast source routing. In other words, the announcement of a node A can be immediately forwarded to node B through intermediate nodes of the same network island (in case A and B are not neighbors). And in consequence, messages could be exchanged using k-hop non caching relay points. The number of hops is limited and does not span all the network island.

4.2.7 Frugal use of the wireless medium

The protocol is designed so as to maximize the message delivery ratio while remaining very frugal as far as the number and the volume of exchanged data are concerned. A node adapts its catalog announcements to the interest profiles of its neighboring nodes. Therefore the periodic announcements do not advertise all the cached messages but only those that might interest the neighbors. Many requests for a message from a source node are all satisfied using only one message broadcast. The protocol is built over UDP, and each node is listening on the broadcast channel.

4.3 Publish/Subscribe interface

The communication layer is accessible to the service layer's agents through a publish/subscribe interface. Service agents can create, publish, subscribe to, and cancel messages.

4.3.1 Message

The communication protocol operates with structured pieces of information referred to as "messages". A message is composed of two parts: a header, and a payload. The header is a collection of attributes, which can provide any kind of information about the corresponding message, such as its origin, its topic, a list of keywords, the type of its content, etc. The payload can be any type of content created by a service agent.

The interface enables the service layer's agents to directly create and manipulate messages. Messages are meant to be published by source nodes, so that they get transported and received by interested nodes. The protocol requires unique identification for each message in the network. A source node sets the message's id using a hashing function. The protocol also requires a source node to set the message's deadline, after which the message will no longer be valid.

Figure 4.1 shows some message examples. The `id` and `deadline` attributes are the only ones required in every message. Other attributes can be added by the creator of the message, for example the `publisher`, `keywords`, and `mimetype` attributes are added by a higher-layer agent as required by its proper protocols.

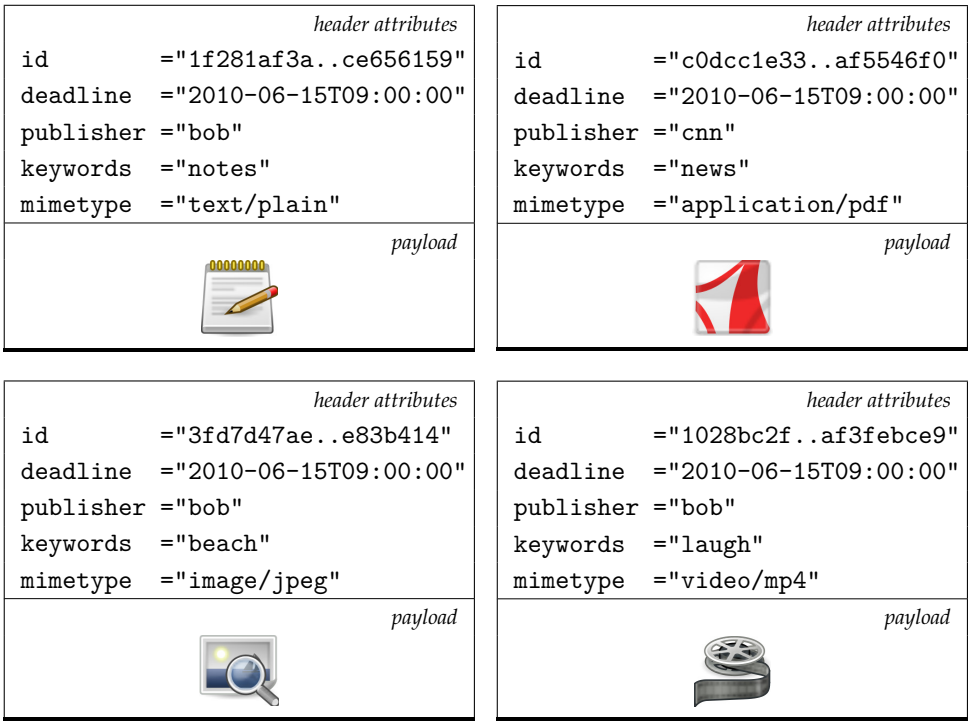


Figure 4.1: Message examples

The payload of a message is compressed to take the least possible space during transmission from node to node. Large messages are split into smaller fragments, a message fragment has the same header as its parent message and a part of the compressed payload.

In order to illustrate the publish/subscribe mechanism described in the following sections, Figure 4.2 presents a communication example between two neighboring nodes *A* and *B*. Node *A* publishes a message, and node *B* subscribes and receives the published message. The example also shows the announce-request-broadcast interactions between the nodes.

4.3.2 Publishing a message

A service layer agent uses the interface to publish a message into the network. The publish method simply deposits the message in the local cache. As explained in Section 4.2, one of the important elements of the protocol is to construct a “catalog" for the mobile node. The catalog lists the messages available in the node’s local cache that can be interesting to the neighboring nodes. The protocol advertises this catalog to these neighboring nodes. If the message can interest a neighbor node, then the message’s header is appended to the catalog. Therefore the message gets advertised. When a node that has subscribed for this kind of message receives the catalog advertisement, this node can request to get the message. The message is therefore transmitted from the publishing node to an interested node, and by the same mechanism the message will also be transmitted

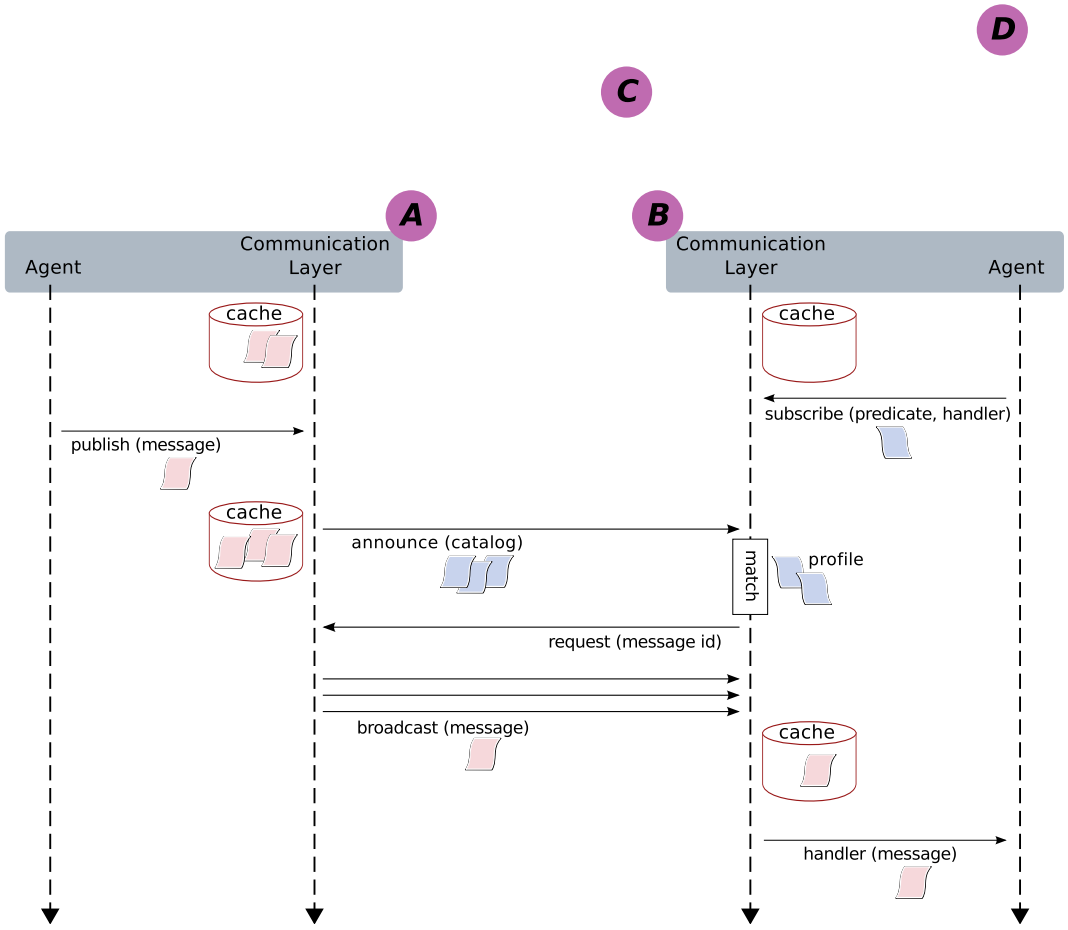


Figure 4.2: Publish/Subscribe interface and subsequent communications between two neighbor nodes

from interested node to other interested nodes.

4.3.3 Subscribing to messages

A service layer agent specifies what kind of messages it wishes to receive from the network. It creates an atomic proposition for every needed attribute comparison, then creates a subscription predicate as a function of these atomic propositions. The subscription predicate is then used at the subscribe method, which appends this predicate to the global node's "profile". The profile is another important element of the protocol (Section 4.2), where all the wishes of the node are grouped. Figure 4.3 shows all the elements of a subscription.

The protocol matches an atomic proposition against corresponding header attributes from all accessible messages (advertised and made accessible by neighboring nodes). In an atomic proposition, the value can be a number to match against numerical attributes, or it can be a regular expression to match against string attributes. The predicate example in Figure 4.4 conveys wishes of receiving french news messages. The second message

$a_i := (attribute, comparator, value)$ is an atomic proposition,
where $comparator \in \{=, \neq, <, \leq, >, \geq\}$
$p_i = a_1 \circ ... \; a_i \; ... \circ a_n$ is a subscription predicate,
where operator $\circ \in \{\wedge, \vee\}$
$P = (p_1 \vee ... \; p_i \; ... \vee p_n)$ is the resulting node profile.

Figure 4.3: Subscription elements

in the above example Figure 4.1 matches this predicate, so when this message becomes accessible it will be requested by the subscribing node.

$p =$ `keywords="news|journal"` $\wedge$ (`publisher="cnn"` $\vee$ `language="fr"`)

Figure 4.4: Predicate example

The subscribe method takes two parameters: first a predicate, and second a handler to notify the service agent of every message reception.

4.3.4 Canceling a message

A service agent may need to cancel an obsolete message before the message's proper deadline, which is highly beneficial from a network load reduction point of view (by stopping the message's dissemination). The interface allows a higher-level agent to cancel the dissemination of messages. The cancel method takes the id of the message to cancel, and a deadline date. A cancel starts purging the message in the local cache, and then it adds an entry to the node's catalog. Therefore, catalog advertisements disseminate the cancellation to neighboring nodes, until the deadline date.

4.4 Discussion

4.4.1 Communication delay

The store-carry-and-forward nature of the protocol imposes communication delays in message delivery. A message usually travels from a source node to another node using an unknown number of intermediate carriers. Communication delay depends on many variables of the mobile environment, for example let's take the distance separating two network nodes:

- If nodes *A* and *B* are neighbors (mutual wireless radio range), the communication delay d_1 depends on the announcement period of both nodes (profile+catalog announcement) added to the request transmission and then to the actual message broadcast time.

- If nodes *A* and *C* are in the same network island but not in wireless range. The communication delay d_2 depends on the period of k-hop announcements and k-hop requesting and proper message transmission times, and on the number n of hops between the nodes (when n>k).
- If nodes *A* and *D* are in separate network islands, then the communication delay d_3 depends on the mobility of intermediate carrier nodes traveling from an island to another in addition to delays inside of islands. The delay caused by the nodes traveling between islands is clearly higher (by many orders of magnitude) to the delays inside of islands.

In consequence, the communication layer provides a best effort solution where the success of an interaction between remote service entities is not guaranteed, and where communications include significant and unpredictable delays. Still, the communication layer provides a network-wide solution in disconnected MANETs, where higher-level applications should consider communication delays as part of their interaction model.

4.4.2 Loose coupling benefits

The communication challenges identified in Chapter 3 show that in order for the service platform to tolerate communication disruptions, the communication layer must enable the decoupling at the three time-synchronization-space dimensions.

- Time: With unpredictable availability there is a high probability that two service entities wanting to communicate are not active at the same time. As seen in the example of Figure 4.2, when node *B* subscribed, the message was not even published yet, as if node *A* was unavailable at that moment.
- Synchronization: In the example, node *A* was not blocked after producing its message, and likewise node *B* was not blocked while waiting to receive the message.
- Addressing space: The two interacting nodes do not need to know and hold references to each other. The producer of information does not know how many are consuming this information. Likewise, the consumer of information does not know how many producers it is getting its information from. Therefore, the content-based protocol can achieve this space decoupling.

As a consequence, the communication challenges of time-synchronization-space decoupling are well met by the used communication protocol.

4.5 Summary

This chapter presented the communication layer of the service platform. Due to the high communication constraints imposed by disconnected environments, I used an opportunistic and content-driven protocol (DoDWAN). The chapter described the inner workings of the protocol, consisting of a message store-carry-and-forward paradigm, content-based matching, and opportunistic gossiping. The chapter also described the publish/subscribe module interfacing the protocol with the upper service layer.

My friend, thou art not my friend, but how shall I make thee understand? My path is not thy path, yet together we talk, hand in hand.

Kahlil Gibran

5 Service Discovery

Contents

5.1 Introduction 41
5.2 Service nodes in mobile environments 42
5.3 Overview 44
5.4 Elements of the discovery protocol 46
5.4.1 Description 46
5.4.2 Advertisement 48
5.4.3 Process at the provider side 52
5.4.4 Collection 53
5.4.5 Selection 53
5.4.6 Process at the client side 54
5.5 Discussion 54
5.6 Summary 55

5.1 Introduction

By definition, discovery enables network nodes to become aware of the availability and capability of peers on the network. Service providers describe and advertise their capabilities for clients to discover and match their needs.

According to the service-oriented paradigm, a service's functional and non-functional properties are described in a machine interpretable and human readable syntax forming the service's contract. The contract is advertised by a service provider on a directory entity. Depending on the characteristics of the network environment, the directory entity may be implemented as centralized, or distributed, or in a peer-to-peer manner. In order for a client to discover services from the directory, the client describes its needs in a format resembling that of the service description, and searches the directory for services matching these needs. The directory may return many matching services to the client, who finally selects one for invocation.

The goal of the proposed service platform is to provide network-wide service provision. Service provision must handle communication constraints inherent in the disconnected mobile environments, where node density and mobility produce fragmented

network topologies (as illustrated in Figure 1.2 on page 3). Service client and provider agents interact using the communication layer (DoDWAN protocol) described in Chapter 4; communications are carried out using best-effort solutions with a content-based store-carry-and-forward protocol. Communications present unpredictable delay times for end-to-end data delivery. Service protocols between providers and consumers must deal with this communication delay in addition to dealing with change and heterogeneity. Therefore, throughout the design of service discovery and invocation, I focus on service agents independence and loosening mutual ties in order to counterbalance the environmental constraints. The goal is to design a service provision platform that is capable of achieving an acceptable level of performance by reducing the completion time of a task as well as reducing the number of needed interactions as much as possible.

This chapter details the discovery protocol of the service layer DiSWAN. Section 5.2 starts by describing the client and provider nodes and their life-cycles. Section 5.3 gives an overview of the architecture. Section 5.4 describes the elements that construct the discovery protocol, from the provider's service description and advertisement until the client's service collection and selection. Section 5.5 discusses the changes that can affect the performance of the service provision process. And Section 5.6 concludes the chapter.

5.2 Service nodes in mobile environments

In a mobile network environment, each node hosts a service platform instance composed of two layers: a service layer called DiSWAN and a communication layer called DoDWAN. Every node starts its life as a simple node with no service agents. A service agent is either a client agent or a provider agent. A provider agent provides one business service, and a client agent wants to use one business service. A node having one or several client agents is called a client node, and a node having one or several provider agents is called a provider node. A simple node becomes a client node or a provider node or both, according to its user's requirements. It can have many client agents and many provider agents.

This work does not address the entire software deployment issues. However a description of a client's and a provider's lifecycle is fundamental in defining their roles in the discovery and invocation protocols.

Provider node's lifecycle.
When the user installs a first provider agent, the simple node becomes a provider node. The user can install many provider agents. The node becomes simple again when the user uninstalls the last provider agent. When an installed provider agent advertises its descriptor, it becomes ready to receive and then respond to client invocations. The advertised descriptor contains functional and non-functional service properties in addition to the node's context properties. If the context of the provider node changes (e.g. changes location), the provider agent updates and advertises its descriptor again. Figure 5.1 shows the lifecycle of a provider node and the lifecycle of an installed provider agent.

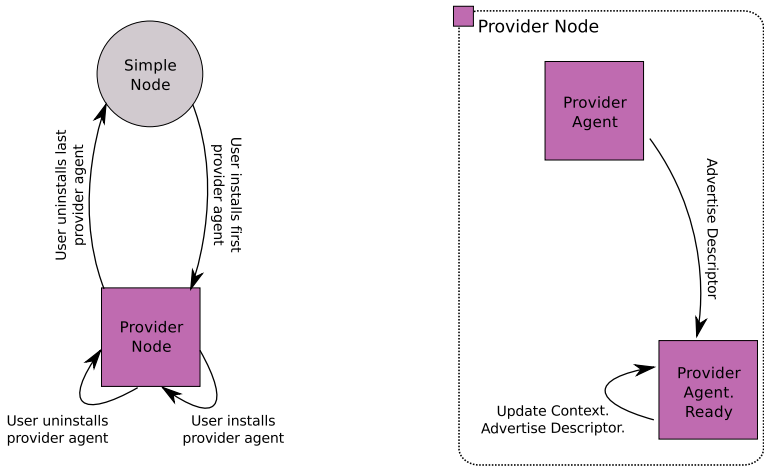


Figure 5.1: Lifecycle of a provider node and of a provider agent

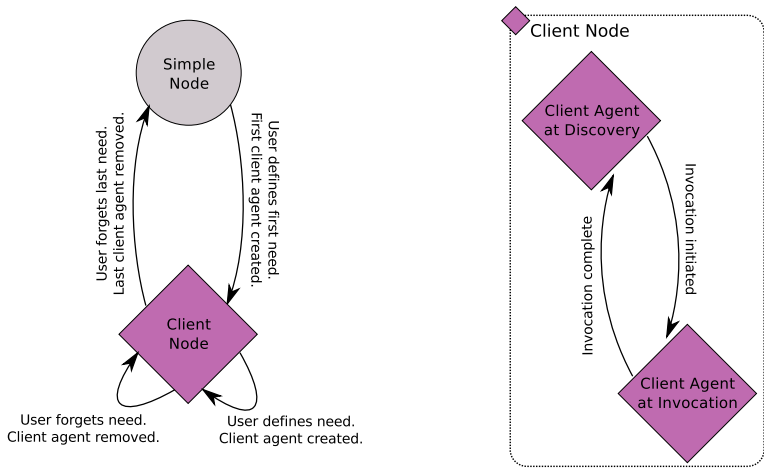


Figure 5.2: Lifecycle of a client node and of a client agent

Client node's lifecycle.

When the user defines discovery needs, the platform creates a corresponding client agent. Discovery needs represent the kind of functional service capabilities the user wants to use, in addition to context properties about where and when he wants to use them. When the user forgets his discovery needs, the corresponding client agent is removed. When the first client agent is created, the simple node becomes a client node. Many client agents can be created, each corresponding to a different set of discovery needs. A node becomes simple again when the last client agent is removed. The client agent starts its life at a discovery stage. It discovers providers capable of fulfilling the user's needs. When the user decides to invoke a discovered provider, the client agent goes into an invocation stage. When the invocation completes, the client agent goes back to discovery stage. In case the user is satisfied with invocation, he may want to forget his discovery needs, so the corresponding client agent is removed. Figure 5.2 shows the lifecycle of a client node and the lifecycle of a client agent.

In addition to the previously described lifecycle, a node can be a client and provider

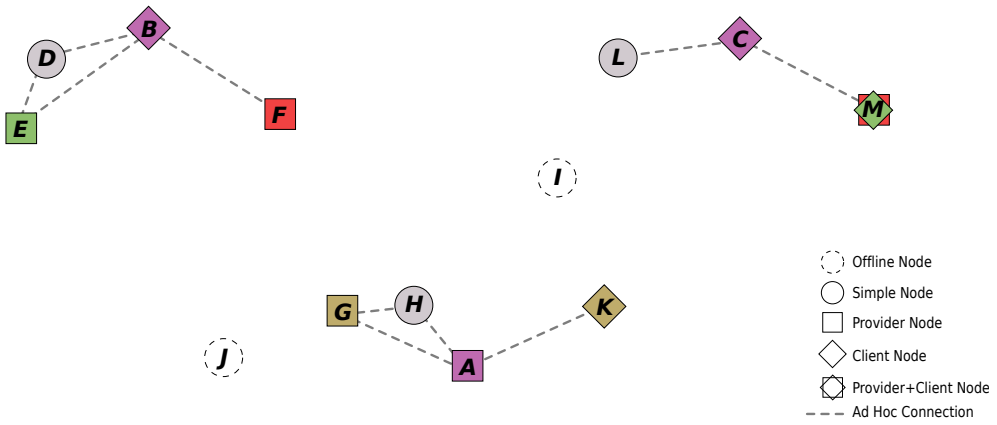


Figure 5.3: Network environment

node if it contains client and provider agents at the same time. Figure 5.3 depicts an example of a mobile environment where nodes form disconnected islands of connectivity. This environment contains client nodes (*B, C, K, M*), provider nodes (*A, E, F, G, M*), simple nodes that are neither clients nor providers (*D, H, L*), and offline nodes (*I, J*). Notice that node *M* is both client and provider.

5.3 Overview

The fundamental steps of the service provision process are: (1) description, (2) advertisement, (3) discovery, and (4) invocation. A service directory is the service entity where providers advertise their services and clients discover needed services.

Directory architecture.
A centralized directory is excluded in MANETs because it is not accessible from all network nodes and it represents a single point of failure. Some architectures distribute the contents of the directory over an overlay network consisting of uniformly deployed directory nodes, where the directory is either fully or partially duplicated. Overlay networks are hard to maintain in highly dynamic and disconnected MANETs. On the other hand, the peer-to-peer architecture is better suited for disconnected environments, where each network node is responsible for its own list of discovered services. But existent implementations of this peer-to-peer architecture have the disadvantage of producing high network traffic, since providers periodically broadcast or multicast their descriptions that are also disseminated to all network nodes (see directory architecture at page 25). Therefore, the solution I propose in the service platform makes every client node responsible for its own local directory as in peer-to-peer architectures, yet using the publish/subscribe content-based opportunistic communications of the underlying layer for discovery interactions. In consequence, my solution gives greater independence for service nodes, and also uses the communication layer's selective information dissemination in order to maintain a low network overhead.

To show how local directories fit into the service architecture, lets take a simple ex-

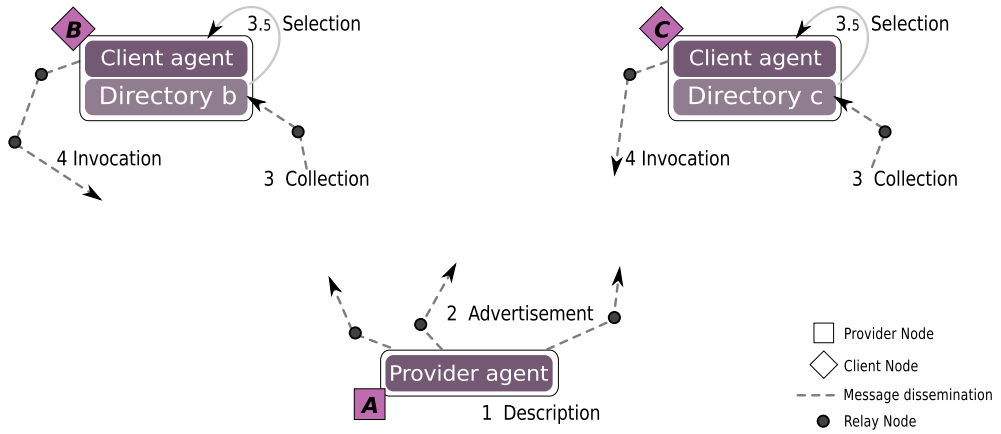


Figure 5.4: Service interactions between a provider node *A* and client nodes *B* and *C*

ample. Consider that the node *A* of Figure 5.3 is providing a service that nodes *B* and *C* want to use. Figure 5.4 depicts the steps of the provision process. (1) The provider creates a description of functional and non-functional service properties in addition to the provider node's context properties. (2) A provider does not need registration in a specific directory, instead it advertises its service descriptor to whoever is interested. A service descriptor is created as a message containing the service description and context properties. The descriptor message is published using the underlying communication layer, so to get disseminated by network nodes until delivered to interested client nodes. (3) Discovery at the client side is divided into collection and selection. A client node is responsible for its own local directory, which contains only interesting services according to its user's needs. The client agent subscribes to interesting service descriptors, descriptors are received over time and collected in the local directory. When the user needs to actually invoke a service, he selects a provider out of the local directory. Therefore, there is no need for a client agent to contact a remote directory as in centralized architectures, and no need to synchronize directory copies as in distributed architectures. (4) The client agent finally invokes the selected provider.

Client's invocation behavior.

The default behavior of a client is first to define its needs. These needs are used to collect interesting service descriptors. This collection process is executed in the background. When the client needs to invoke a provider, it selects one from the supposedly collected descriptors.

- In case the selection succeeds, then the client is able to create a well formatted invocation request according to the selected descriptor. The classic invocation behavior would be to send this request specifically to the discovered provider. In the invocation solution I propose (detailed in Chapter 6): the invocation request formatted according to a discovered provider, can be received (using a content-based request) by providers other than the discovered provider, as long as these other providers offer a compatible business service. This proposed behavior extends the classic one, it uses the discovery stage as means to invoke many compatible providers possibly available in the environment. By extending the invocation reach to other compati-

ble providers, the degree of compatibility influences the reliability of the responses from such providers.

- In case the selection did not succeed (e.g. no collected descriptors yet), and the client needs an imminent invocation, then a content-based invocation request can be created by the client without referring to any known provider. This behavior (also detailed in Chapter 6) is a last minute attempt to invoke some provider. But since no actual description about a possibly responding provider was collected, then the response of such a provider should not be considered as reliable in comparison with the response of a discovered provider.

5.4 Elements of the discovery protocol

The purpose of the proposed discovery protocol is to let every client find an interesting service provider. The protocol works in a typical scenario, where a client node and a provider node are considered distant with no temporaneous end-to-end connection. The same protocol remains valid in scenarios where client and provider nodes are in proximity. The discovery protocol consists of all the provision steps needed before an invocation, which are:

- *Description* and *advertisement* are performed by the service provider.
- *Collection* and *selection* are performed by the service client.

5.4.1 Description

When the user of a node installs a service, DiSWAN creates a provider agent. The provider agent envelopes the description of functional and non-functional properties along with the provider node's context properties in a single message called service descriptor. Therefore, a descriptor is essential since it contains all the resource information needed by consuming clients to decide whether or not the provider fulfills their needs. The main components of a service descriptor are (see Figure 5.5): (1) the provider node's identity, (2) the service's identity, (3) a set of the service's functional capabilities, (4) a set of the service's non-functional detail, and (5) a set of the provider node's context properties.

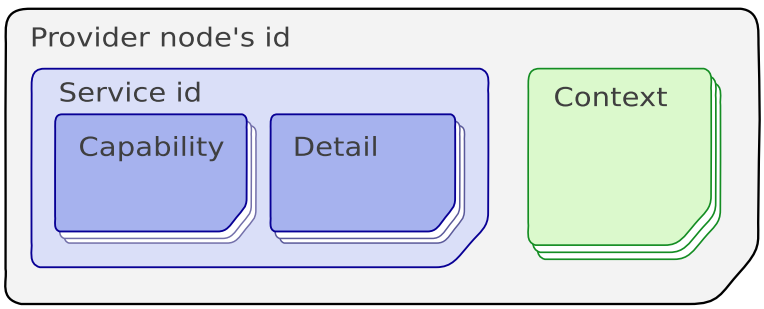


Figure 5.5: Components of a service descriptor message

5.4.1.1 Functional and non-functional service properties

Service description aims to list functional capabilities and non-functional detail. It is the job of the service creator (programmer) to define these functional and non-functional properties. When the service is installed, these properties become available in the corresponding provider agent.

Functional properties form the service interface that a client needs in order to create requests and understand responses. Generally in distributed client/server computing, the functional interface of a server is hard coded in a proxy that has to be installed on the client. This forces an implementation technology on both the service and client sides. Following the ambition to loosely couple the service computing at the implementation level, a description written in a universal syntax gives more independence to the service entities. Therefore I chose to use the WSDL description language (as in Web Services), and not serialization methods proper to a specific implementation technology like it is the case in Jini [72] for example. This way, the client is free to create its own proxy according to its implementation technology. Figure 5.6 shows a functional description of an example service "S1" offering the capability "add" that takes two numbers and returns their sum.

```
<wsdl>
  <interface name="S1">
   <operation name="add">
    <input name="number1" xs:type="xsd:int"/>
    <input name="number2" xs:type="xsd:int"/>
    <output name="result" xs:type="xsd:int"/>
   </operation>
  </interface>
</wsdl>
```

Figure 5.6: Example of functional service properties written in WSDL

Extra detail enhances the service description with non-functional properties (e.g. author, version, cost, signature, etc.). These non-functional properties can be decisive when a user chooses between many providers of the same functionality. Figure 5.7 shows an example of non-functional service properties written in simple XML.

```
<detail>
   <version>2.0</version>
   <author>romeo</author>
</detail>
```

Figure 5.7: Example of non-functional service properties written in XML

5.4.1.2 Provider context properties

Because of the dynamically changing environment (node mobility, volatility, density), a provider node can be available in the environment at a given time, and it can leave the

environment or simply be turned off to become unreachable by client nodes at some other point in time. Therefore, a provider node's user has to convey its context properties in terms of availability in space and time. These context properties describe the provider node's availability using a representation of an iCalendar calendar event, written in XML using xCal[1]. Figure 5.8 shows an example of context properties.

```
<context>
   <vevent>
    <properties>
     <dtstart><date-time>20100612T090000Z</date-time></dtstart>
     <dtend><date-time>20100612T180000Z</date-time></dtend>
     <location>vannes,tohannic,ubs</location>
    </properties>
   </vevent>
</context>
```

Figure 5.8: Example of a provider node's context properties

The `<context>` element contains a calendar `<vevent>` element specifying the availability in time (`<dtstart>` to `<dtend>`) and space (`<location>`). The provider can even convey many availability events, considered as the node's planning. The location property is a general description of the environment: whether it's a university campus, a city's public place, or a building.

5.4.2 Advertisement

After describing the service properties and the provider node's context properties, the description is enveloped in a descriptor that is advertised to interested client nodes. The descriptor is created as a message and published using the underlying communication layer. The XML description is put in the message's payload. The message header contains the message's "`id`" and "`deadline`" that are required by the communication protocol.

Even though the description is intended to be accessible by the higher service layer of a receiving client node, it is not enough to just put all the properties (functional, non-functional, and context) into the payload part of the message because messages get disseminated in the environment according to their header's attributes. Therefore, for a more effective descriptor dissemination and discovery, some properties of the payload need to also be present at the header part of the descriptor message. Figure 5.9 specifically shows that data in the payload is only accessible to the service layer entities and data in the header is accessible to the communication layer. In consequence, if more information about the payload is also present in the header, then the communication layer works more effectively in the selective dissemination of messages. Figure 5.10 shows the descriptor message's header attributes. Attributes with the "`s:`" and "`e:`" prefix are predefined by the DiSWAN service layer.

The values of the attributes are automatically filled according to the description in-

[1]iCalendar (RFC 5545) stands for: Internet Calendaring and Scheduling Core Object Specification, and xCal is an XML representation of iCalendar.

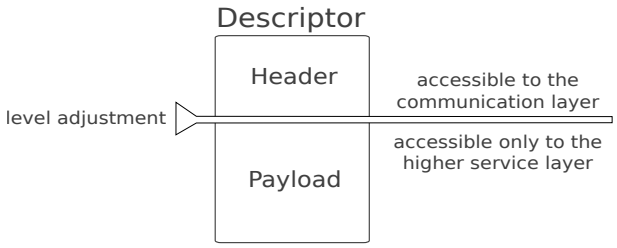


Figure 5.9: Controlling the level of accessible information

	header attributes
`id`	`=""`
`deadline`	`=""`
`publisher`	`=""`
`s:type`	`="descriptor"`
`s:service`	`=""`
`s:capability`	`=""`
`e:detail/[property]`	`=""`
`s:location`	`=""`
`s:dtstart`	`=""`
`s:dtend`	`=""`

Figure 5.10: Header attributes of a descriptor message

formation at the creation time of the provider agent. The header's attributes are the following:

- Attributes required by the communication layer: a required unique message `id` attribute, a required `deadline` attribute to declare the validity of the message.
- A `publisher` attribute (predefined by the communication layer) holds the provider node's identity.
- An `s:type` attribute declares a descriptor message.
- A service identity `s:service` attribute. Its value is the equal to the WSDL interface name.
- A `s:capability` attribute representing functional operations. Its value is a comma separated list of the names of WSDL operations.
- Non-functional service attributes are represented in the header as extensible attributes with the "`e:`" prefix. They use the form `e:detail/[property]`. They are the equivalent to the non-functional properties of the detail description present in the payload. An attribute is added for each element in the detail part (e.g. `e:detail/version` and `e:detail/author`).
- The context attributes are the `s:location` attribute for helping clients to discover services at a specific location, and the `s:dtstart` and `s:dtend` attributes for clients

wishing to discover services at specific times. These attributes are the equivalent to the `<location>`, `<dtstart>`, and `<dtend>` elements of the context properties.

Figure 5.11 shows an example of a complete descriptor message. The message's payload contains the full description as an XML listing, which contains the functional service description in the `<wsdl>` element, the non-functional service description in the `<detail>` element, and the provider node's context properties in the `<context>` element. The header part contains the attributes that are automatically filled according to the full description.

header attributes

```
id                ="af435ce3b23366f0d77cef8bfd6f4f81"
deadline          ="2010-06-15T09:00:00"
publisher         ="bobmobile"
s:type            ="descriptor"
s:service         ="S1"
s:capability      ="add,subtract"
e:detail/version  ="2.0"
e:detail/author   ="romeo"
s:location        ="vannes,tohannic,ubs"
s:dtstart         ="2010-06-12T09:00:00"
s:dtend           ="2010-06-12T18:00:00"
```

payload

```
<wsdl>
  <interface name="S1">
   <operation name="add">
    <input name="number1" xs:type="xsd:int"/>
    <input name="number2" xs:type="xsd:int"/>
    <output name="result" xs:type="xsd:int"/>
   </operation>
   <operation name="subtract">
    <input name="number1" xs:type="xsd:int"/>
    <input name="number2" xs:type="xsd:int"/>
    <output name="result" xs:type="xsd:int"/>
   </operation>
  </interface>
</wsdl>
<detail>
   <version>2.0</version>
   <author>romeo</author>
</detail>
<context>
   <vevent>
    <properties>
     <dtstart><date-time>20100612T090000Z</date-time></dtstart>
     <dtend><date-time>20100612T180000Z</date-time></dtend>
     <location>vannes,tohannic,ubs</location>
    </properties>
   </vevent>
</context>
```

Figure 5.11: Example of a descriptor message

5.4.3 Process at the provider side

Figure 5.12 recapitulates the steps taken at the provider side in order to install a service and get ready for incoming client invocations. A programmer creates the service's binary, functional WSDL description, and non-functional description. A provider agent hosts the service binary, and creates the service descriptor according to the description and context properties. The descriptor is published to interested clients. After advertising the descriptor, the provider agent subscribes for client invocation requests in order to receive them. Chapter 6 details how to create the corresponding subscription predicates.

Service Creation

1 **begin** Programmer creates service S1 and its description
  service binary classes —> `service.binary`
  define functional interface: operations, inputs, outputs —> `service.wsdl`
  define non-functional service details —> `service.detail`

Provider Agent Creation

2 **begin** `pa = new ProviderAgent(S1, node.context); //User installs service S1`
3 `pa.business(S1.binary); //set service binary`
4 **begin** `payload = new Payload(); //create a message payload`
    `payload.append(S1.wsdl); //add WSDL functional interface`
    `payload.append(S1.detail); //add non-functional service details`
    `payload.append(node.context); //add context properties`
5 **begin** `header = new Header(); //create a message header`
    `//repeat fill from equivalent descriptions`
    ...
    `header.setAttribute("s:capability","add");`
    ...
6 `pa.descriptor(new Message(header, payload)); //set descriptor`
7 `publish(pa.descriptor); //publish message in the environment`
8 `p = new Predicate(s:type="request" ∧ destination="bob"); //create predicate for incoming client requests`
9 `h = new Handler(pa); //create a reception notification handler`
10 `subscribe(p, h); //subscribe to client requests`

Provider Agent Runtime

11 **Idle**

notify `h` triggered 12 respond to request	**if** changes occur: updated service description updated provider context then **restart** at step 4

Figure 5.12: Provider side process, from service creation to getting ready for invocation

5.4.4 Collection

When the user defines his discovery needs, the platform creates a client agent. In order for the client node to collect interesting services, the client agent subscribes using a subscription predicate [2]. The predicate contains propositions that are matched against the attributes of a descriptor message.

For example, the client can define his needs of using a service offering a capability named "add" or "sum" and that the provider is located at the specific "ubs" location. The client agent subscribes using the following predicate corresponding to the user's needs:

$$p_1 = \texttt{s:type="descriptor"} \wedge \texttt{s:capability="add|sum"} \wedge \texttt{s:location="ubs"} \tag{5.1}$$

Now if this client wants to restrict its discovery to providers available at a specific time, the client agent subscribes using the following predicate:

$$p_2 = p_1 \wedge \texttt{s:dtstart<"2010-06-12T10:00:00"} \wedge \texttt{s:dtend>"2010-06-12T11:00:00"} \tag{5.2}$$

Effectively, the subscription using the publish/subscribe interface of the communication layer enables the content-based opportunistic matching process between the client's preferences against the descriptors that are disseminated in the network. The matching consists of comparing the attribute-value fields of the header of a disseminated descriptor to the subscription predicate (as was explained in the previous chapter). As time passes, all matching descriptors are collected in the client node's local directory for future use when a service invocation is needed.

5.4.5 Selection

During collection, the platform presents the user with the already collected services in the local directory. At this stage, the client agent can access the information of a descriptor's payload. This enables the client to make sure of a proper functionality match, and a proper contextual availability according to the current time and space situation. Selection is the process of picking a proper service provider in preparation for invocation.

The invocation time can be a high level decision made by the user (human or application). When an invocation is needed, a service descriptor is selected according to the current context properties. Effectively, the client agent compares its node context with the context advertised in the descriptor in order to select the best available provider node. For example, if the client node is present at the "ubs" location at 10am, then the client agent selects a provider that had advertised that it would be present at the "ubs" location and available at the needed time.

The platform also enables the client agent to be notified when the client node's context changes (e.g. location) to match that of a collected descriptor. Therefore, the matching descriptor is selected and a notification is raised to the user in order for him to opportunistically invoke the provider.

[2] Subscription ingredients:
$a_i := (attribute, comparator, value)$ is an atomic proposition, where $comparator \in \{=, >, \geq, <, \leq, \neq\}$, and $p = a_1 \circ ... \; a_i \; ... \circ a_n$ is a subscription predicate, where operator $\circ \in \{\wedge, \vee\}$

5.4.6 Process at the client side

Figure 5.13 recapitulates the process of a client node collecting interesting services, until the invocation time when the user selects a service out of his local directory.

Discovery Preferences

```
1 User defines discovery preferences
```

Client Agent Creation

```
2 begin ca = new ClientAgent(); //create client agent
3   begin p = new Predicate(); //create predicate
      p.add(s:type="descriptor"); //add a descriptor type attribute
      p.and(s:capability="add|sum"); //functional capabilities and detail
      ...
      p.and(s:location="ubs"); //add context properties to restrict the discovery
      p.and(s:dtstart<"2010-06-12T10:00:00");
      p.and(s:dtend>"2010-06-12T11:00:00");
4   begin ca.collect(p);//collect service descriptors
      h = new Handler(ca); //create a reception notification handler
      subscribe(p, h); //subscribe to service descriptors
```

Client Agent Runtime

```
5 Idle
```

notify `h.process(message)` triggered

```
6 begin
    //store collected descriptor
    directory.append(message)
    //create opportunistic context handler
    ch =new ContextHandler(message.context)
  end
```

(**when** invocation needed) **or** (**notify** `ch` triggered)

```
7 begin
    //select descriptor
    d = ca.select(node.context);
    //form functional invocation request
    request =new Request(d, user.input);
    //publish invocation request
    ca.invoke(request);
  end
```

Figure 5.13: Process at the client side

5.5 Discussion

The primary performance criterion of the service provision process is the time delay that a task needs to complete, a task can be composed of up to several round trip interactions between two service entities. As far as discovery is concerned, the proposed discovery protocol described in this chapter relies on the information dissemination properties of the underlying communication layer, which in consequence reduces the discovery interactions to just one (getting the service descriptor from the provider node to the client node). Effectively, the provider node advertises its descriptor by making a publication, and the a client node receives this descriptor by subscribing. Therefore the time delay

of discovery is that of delivering a message from a provider node to an interested client node, this delay is unpredictable because of the dynamic and disconnected mobile environment.

In addition to the communication delays affecting the performance of discovery, there exists other elements that directly influence the performance of the service provision. From a client's point of view, these elements are mostly related to changes that can occur at the provider side. Reasons for this change can be simply related to the service versioning, or in case the provider of the service decides to change its implementation aspects. Changes to the identity or to the interface of the service render the invocation impossible for clients that have not updated their knowledge about the service they seek. Moreover, the unpredictable node availability of service providers in disconnected MANETs and the absence of instantaneous up-to-date information about this availability may render the advertised context properties obsolete. Therefore clients obviously need an additional discovery phase, adding extra delays to the service provision process. In order to reduce the effect of these changes on the performance of service provision, the proposed discovery protocol enables an "always on" discovery. Effectively, a client agent always collects service descriptors in an effort to have the most possible up-to-date information about service providers. Service providers on the other hand advertise their changes in service functionality and changes in context (they publish new updated descriptors). This discovery behavior characterizes the publish/subscribe communication model.

The main design focus in the service platform is to improve the loose coupling between the providers and the consumers of services. The discovery protocol enables a form of loose coupling by reducing interactions between providers and clients to the minimum. Still, the more knowledge about the provider that a client has to remember and to keep up-to-date, the more ties exist between the provider and client entities. When an invocation is needed, it becomes harder to handle a change of availability of the provider, hence reducing the performance of the service provision. The next chapter proposes elaborate invocation solutions in order to increase loose coupling and enhance the service provision performance. More precisely, content-based invocation requests enable the invocation of multiple providers, therefore reducing the effect of obsolete context properties for example.

5.6 Summary

This chapter presented the behavior of the proposed discovery protocol in DiSWAN. It presented the life-cycles of the client and provider nodes. A node is a client node if it hosts a client agent, the client agent is responsible for discovering service descriptors according to its user needs and also responsible for invocation interactions with a service provider. A node is a provider node if it hosts a provider agent, the provider agent is responsible for exposing a business service and also responsible for advertising the service's descriptor. The chapter also described the discovery architecture using local directories, where each client node is responsible for its own local directory where it collects interesting service descriptors. The chapter detailed the elements that construct the discovery protocol, they are the description and advertisement at the provider side, and collection and selection at the client side. Description is the process of describing the

functional and non-functional service properties in addition to describing the provider node's context properties. Advertisement is the process of creating a service descriptor and publishing it to interested clients. Collection is the process of describing a client's needs and subscribing using these needs in order to receive matching service descriptors. Selection is the process of selecting a proper service descriptor from the local directory in preparation for invocation. Invocation is detailed in the following Chapter 6.

6 Service Invocation Solutions

Contents

6.1 Introduction . . . 57
6.2 Service provider redundancy . . . 58
6.3 Remote invocation . . . 60
6.3.1 How clients issue invocation requests to providers . . . 60
6.3.2 How providers respond to client requests . . . 62
6.3.3 Response management policy . . . 66
6.3.4 Network healing . . . 67
6.3.5 Discussion . . . 71
6.4 Invocation restrictions . . . 73
6.5 Blind invocation: Bypassing discovery . . . 75
6.6 Client and provider states . . . 76
6.7 Remote invocations of stateful services . . . 77
6.8 Public invocations . . . 80
6.9 Perspectives . . . 81
6.9.1 Semantic invocation . . . 81
6.9.2 Complex request . . . 81
6.10 Summary . . . 82

6.1 Introduction

Service invocation is the process of a client actually using the capabilities of a service provider. In the previous chapter, I described the service discovery process. This chapter describes the full solutions that I put into place in the DiSWAN service layer in order to handle service invocations. Like discovery, invocations must also be tolerant to communication delays that are naturally induced by disconnected network environments. The amount of delay that a client can tolerate depends on the interactivity level needed (at business level) for reasonably good operating conditions. From a client's point of view, after issuing its request, it may expect one or many responses. According to these client

expectations, service providers should know what and how to deliver their services. The presented solutions use remote invocations only, they do not deal with code mobility [1].

Section 6.2 presents how some providers in a mobile environment might be offering the same business service. Section 6.3 presents the default remote invocation solution proposed in DiSWAN, detailing how a client can issue a content-based invocation request to all providers of a business service, and how these providers respond back. The section also presents network healing techniques that are use to eliminate unneeded invocation messages. Section 6.4 presents how a client uses invocation restrictions in order to address specific providers. Section 6.5 presents how a client can even send a blind invocation request in case of a discovery failure. Section 6.6 presents the client and provider states during the provision process. Section 6.7 details stateful remote invocations via session management. Section 6.8 details public invocations where a response can benefit more than one client. Section 6.9 presents some perspectives. And Section 6.10 concludes the chapter.

6.2 Service provider redundancy

Mobile networks can be perceived as environments containing many providers offering a wide variety of services, where it becomes likely to have some providers offering the same business service. This situation is the result of having different providers host the same service implementation, or providers hosting different service implementations with the same declared invocation interface, or providers hosting different service implementations with different but equivalent invocation interfaces. Let's take the comparison example given in Figure 6.1:

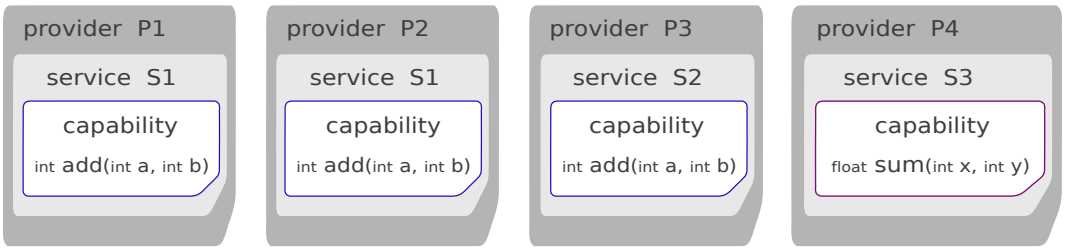


Figure 6.1: Provider examples offering the same business service

1. Providers P1 and P2 have installed the same service S1 so they are naturally offering the same capability `add(a,b)`.
2. Provider P3 has installed service S2 which happens to offer the capability `add(a,b)`, same as offered by service S1.
3. Provider P4 has installed service S3 which offers the capability `sum(x,y)`. Capabilities `sum(x,y)` and `add(a,b)` might be equivalent, therefore P4 probably offers the

[1] Code Mobility: First is the transfer of program code (i.e., passing executable code to the target system). Second, one must restore the execution state at the new system; of course, this implies that the execution state was safely backed up at the originating system. Third, all resources or data associated with the old system must be propagated to the target system.

same business service as the rest of the providers.

Increased service availability.
The availability of a business service has a substantial effect on the success and performance of service provision. The more this business service is available in time and in space, the better opportunities there are for the invoking clients. Availability in time depends on the hosting node's behavior (up times), which is assumed to be unpredictable if not advertised. Likewise, availability in space depends on the hosting node's behavior (unpredictable mobility), but also depends on the number of services offering the same capability. Figure 6.1 is an example of how the same business service can be offered by many providers; invocation solutions detailed in the remainder of this chapter take advantage of the improved service availability.

Better service provision performance.
The performance of the service provision process is mainly judged by the time elapsed from the point when a client needs a service to the point when this client gets satisfied, the less time the provision process gets delayed means better performance. Since a message delivery delay from endpoint to endpoint is unpredictable then the amelioration to the provision process has to deal with decreasing the number of client-provider interactions in order for the provision to complete. With that goal in mind, having many providers seamlessly offering the same business service and substituting each other when it comes to responding to a client request provides faster responses for two reasons:

- Failure in client-provider communications: a failure in communicating a request or a response until the invocation deadline, forces the client to redo the invocation. Failures occur in case the originally intended provider has become unreachable or unavailable.
- Message dissemination: a request message is disseminated from a client node and gets processed by the first compatible provider node it reaches, because the final destination of an invocation request is not restricted to a specific provider node anymore.

Service group.
A service "group" is a set of compatible providers offering the same business service (same capability). I use this definition through the remaining of this chapter. A group does not mean any special formation or affiliation, but only a way to denote compatible providers.

The problem.
The point is how clients can take advantage of this provider redundancy with no rediscovery phase. The solution I propose in the following sections is: a client should have the possibility of invoking the capability itself without any regard to the hosting provider, providers on the other hand have to recognize client invocations.

In a wireless ad hoc environment containing a wide range of applications, each service client has invocation needs that differ from other clients. A client's needs imply

how providers handle invocation interactions. Furthermore, having redundant providers does not necessarily produce redundant responses. Depending on the application, if each response depends on its provider's context then a client may need to receive many responses and perform some post processing. For example, providers of an ambient temperature service may produce responses different from each other.

6.3 Remote invocation

A client consumes a service by invoking its capabilities. A business service may be offered by multiple redundant providers. Therefore the default remote invocation behavior in DiSWAN consists of the following:

- A client creates a functional request according to a discovered service provider.
- The client publishes its request message to the attention of the discovered provider in addition to other compatible providers.
- Compatible providers can receive the client's request. A provider creates a response message and publishes it back to the client.
- According to the client's needs, it may get satisfied when receiving the first response or it may collect multiple responses to be satisfied. The invocation completes when the client gets satisfied.
- After being satisfied, the client can initiate the healing of residual request and response messages from the network. The healing of response messages can also be initiated by the providers themselves according to the priority of the response messages.

6.3.1 How clients issue invocation requests to providers

When an invocation is needed, the client selects a discovered service provider. The client agent creates a SOAP request according to the WSDL description of the selected provider. The client agent creates a request message and puts the SOAP request in the payload. The request message's header contains the attributes needed to disseminate to the business service that is hosted by the discovered provider and by other providers. In order to let the request be received by other providers, the service identity and the invoked capability itself are included in the request's header. Figure 6.2 shows a request message's header, some header attributes are inherited from the underlying communication protocol (DoDWAN), the "s:" prefix denotes attributes proper to the service layer (DiSWAN). The header's attributes are:

- The inherited `id` attribute ensures the uniqueness of the message in the network.
- The `s:type` attribute defines the message as an invocation request.
- The `s:issuedAt` attribute is the request creation date.
- Using the inherited `deadline`, the client informs the network of the date it needs to receive a response. The deadline time constraint sets the validity of the request and its corresponding response. The request message is sent with a deadline property

	header attributes
id	=""
s:type	="request"
s:issuedAt	=""
deadline	=""
publisher	=""
destination	=""
s:service	=""
s:capability	=""
s:resPolicy	=""
s:healing	=""

Figure 6.2: Invocation request's header attributes

to inform the network of the validity of this request, a corresponding response has also the same deadline property. This time constraint will guarantee (guaranteed by the communication layer) that only valid requests and responses are present in the network at a given time. In fact, any network node will delete an obsolete request or response from its communication cache and will no longer relay it. I identified three parameters influencing how the client agent defines the deadline: An application related delay constraint, before which the client application needs to receive a response in order to perform other processing; a mobility related delay constraint, if the mobile node knows that it will be leaving its current network environment at a certain time, then it should fix the deadline accordingly; a communication related delay *RTT*, that is a constraint of the network environment, *RTT* is the time needed by a message to go round trip between two network nodes. Therefore the `deadline` attribute is defined by the client according to the following interval:

$$RTT < \texttt{deadline} \leq min(D_{application}, D_{mobility}) \tag{6.1}$$

Using the deadline attribute, the client agent declares the time delay it is willing to wait before receiving a response. Outside of the interval, the invocation cycle is highly prone to failure. The *RTT* is unpredictable in disconnected MANETs, if it exceeds the deadline then the response would not be received by the needed date, therefore the client agent assumes a failed invocation.

- The inherited `publisher` attribute contains the requesting node's identity, so that the response can find its way back to the client.
- The inherited `destination` attribute defines the provider node's identity.
- The `s:service` attribute defines the service identity as described in the selected service description.
- The `s:capability` attribute holds the name of the service's capability.
- The `s:resPolicy` attribute declares the response management policy needed by the client . Its value is either `"multiple"` to declare a *multiple-response* policy or `"first"` to declare a *first-response* policy.

- The `s:healing` attribute declares the type of healing that the client needs . Its value is either `"safe"` or `"aggressive"`.

Figure 6.3 shows a request message example. The payload contains the well formatted SOAP invocation. The header contains the discovered provider node's identity `P1`, the service identity `S1`, and the invoked capability `add`.

header attributes

```
id            ="b89f6db2b9515b88cc93b7ee38ca870d"
s:type        ="request"
s:issuedAt    ="2010-06-14T09:00:00"
deadline      ="2010-06-15T09:00:00"
publisher     ="alicemobile"
destination   ="P1"
s:service     ="S1"
s:capability  ="add"
s:resPolicy   ="first"
s:healing     ="aggressive"
```

payload

```
<SOAP-ENV:Body>
 <add>
   <number1 xsi:type='xsd:int'>965654</number>
    <number2 xsi:type='xsd:int'>67622</number>
 </add>
</SOAP-ENV:Body>
```

Figure 6.3: Request message example

6.3.2 How providers respond to client requests

In the default invocation, a client does not orchestrate invocations by sending one request per provider. A client invokes the discovered provider as well as other compatible providers by publishing a single request. Effectively, the request's header illustrated in Figure 6.2 contains the `destination` attribute of the discovered provider, in addition to the `s:service` and `s:capability` attributes enabling the request to be disseminated to compatible providers. In order for providers to receive the request message, each provider P subscribes using a subscription predicate [2] of the following form:

$$p_{\text{P}} = \texttt{s:type="request"} \wedge \big(\texttt{destination="P"} \vee \texttt{s:service=""} \vee \texttt{s:capability=""}\big) \tag{6.2}$$

For example, P1 subscribes using the predicate:

$$p_{\text{P1}} = \texttt{s:type="request"} \wedge \big(\texttt{destination="P1"} \vee \texttt{s:service="S1"} \vee \texttt{s:capability="add"}\big) \tag{6.3}$$

[2]Subscription ingredients:
$a_i := (attribute, comparator, value)$ is an atomic proposition, where $comparator \in \{=, >, \geq, <, \leq, \neq\}$, and $p = a_1 \circ ... \, a_i \, ... \circ a_n$ is a subscription predicate, where operator $\circ \in \{\wedge, \vee\}$

Likewise, each of P2 and P3 uses its appropriate predicate:

$$p_{\text{P2}} = \texttt{s:type="request"} \wedge \big(\texttt{destination="P2"} \vee \texttt{s:service="S1"} \vee \texttt{s:capability="add"}\big) \tag{6.4}$$

$$p_{\text{P3}} = \texttt{s:type="request"} \wedge \big(\texttt{destination="P3"} \vee \texttt{s:service="S2"} \vee \texttt{s:capability="add"}\big) \tag{6.5}$$

Provider P4 on the other hand offers the `sum` capability. For provider P4 to be able to receive the client's request, the `add` capability must be appended next to the proper `sum` capability in the predicate:

$$p_{\text{P4}} = \texttt{s:type="request"} \wedge \big(\texttt{destination="P4"} \vee \texttt{s:service="S3"} \vee \texttt{s:capability="sum|add"}\big) \tag{6.6}$$

It is the user that decides —at the creation time of the provider agent at P4— that `add` represents an alias to `sum` and therefore it is willing to accept the corresponding requests. Still there is a probability that this assumption is wrong, therefore the response of P4 should not be considered as reliable as P1's response for example.

Figure 6.4 shows the header of a response message. The header's attributes are the following:

	header attributes
id	=""
s:type	="response"
s:issuedAt	=""
deadline	=""
publisher	=""
destination	=""
s:service	=""
s:capability	=""
s:requestID	=""
s:requestAt	=""
s:resPolicy	=""
s:healing	=""
s:doubt	=""

Figure 6.4: Invocation response's header attributes

- The inherited `id` attribute insures the uniqueness of the message in the network.
- The `s:type` attribute defines the message as an invocation response.
- The `s:issuedAt` attribute is the response creation date.
- The inherited `deadline` attribute ensures the message will be purged at a future time. It is equal to the request's deadline attribute, since it was supposed to set the validity of the request and its corresponding response.
- The inherited `publisher` attribute contains the responding node's identity.
- The inherited `destination` attribute contains the client node's identity.

- The `s:service` attribute contains the service identity as described in the service description.
- The `s:capability` attribute contains the name of the service's capability.
- The `s:requestID` attribute contains the request's ID.
- The `s:requestAt` attribute contains the request's creation date.
- The `s:doubt` attribute contains the doubt factor of the responding provider. This attribute defines the reliability of the response. If a provider receives a request with a `destination` attribute matching its identity then the provider responds with a doubt of "0" (since it is the discovered provider itself). If a provider receives a request with an `s:service` attribute matching its offered service identity then the provider responds with a doubt of "0" (since a response from an matching service implementation is also considered as fully reliable). If a provider receives a request that does not match neither of the provider identity nor the service identity but only matching the `s:capability` attribute then the response must not be considered as reliable. Therefore the provider puts a doubt of "1" if it offers the requested capability, or puts a doubt of "2" if it offers an alias to the requested capability.
- The `s:resPolicy` attribute contains the same value contained in the corresponding request message (either `"multiple"` or `"first"`).
- The `s:healing` attribute contains the same value contained in the corresponding request message (either `"safe"` or `"aggressive"`).
- The payload contains the well formatted SOAP invocation response.

Figure 6.5 shows response messages from the four providers of the presented example in case they receive the request of Figure 6.3. Both P1 and P2 have a doubt factor of "0", P3 has a doubt factor of "1" since it used capability matching, and P4 has a doubt factor of "2" since it used capability alias matching.

header attributes

```
id            ="e57047912..6b1c2a65"
s:type        ="response"
s:issuedAt    ="2010-06-14T09:10:00"
deadline      ="2010-06-15T09:00:00"
publisher     ="P1"
destination   ="alicemobile"
s:service     ="S1"
s:capability  ="add"
s:requestID   ="b89f6db2..38ca870d"
s:requestAt   ="2010-06-14T09:00:00"
s:resPolicy   ="first"
s:healing     ="aggressive"
s:doubt       ="0"
```

payload

```
<SOAP-ENV:Body> <add>
 <result xsi:type='xsd:int'>1033276
</result> </add>
</SOAP-ENV:Body>
```

(a) P1's Response Message

header attributes

```
id            ="69ae4c15d..45fb1c1a"
s:type        ="response"
s:issuedAt    ="2010-06-14T09:07:00"
deadline      ="2010-06-15T09:00:00"
publisher     ="P2"
destination   ="alicemobile"
s:service     ="S1"
s:capability  ="add"
s:requestID   ="b89f6db2..38ca870d"
s:requestAt   ="2010-06-14T09:00:00"
s:resPolicy   ="first"
s:healing     ="aggressive"
s:doubt       ="0"
```

payload

```
<SOAP-ENV:Body> <add>
 <result xsi:type='xsd:int'>1033276
</result> </add>
</SOAP-ENV:Body>
```

(b) P2's Response Message

header attributes

```
id            ="7db944f96..9a276089"
s:type        ="response"
s:issuedAt    ="2010-06-14T09:15:00"
deadline      ="2010-06-15T09:00:00"
publisher     ="P3"
destination   ="alicemobile"
s:service     ="S2"
s:capability  ="add"
s:requestID   ="b89f6db2..38ca870d"
s:requestAt   ="2010-06-14T09:00:00"
s:resPolicy   ="first"
s:healing     ="aggressive"
s:doubt       ="1"
```

payload

```
<SOAP-ENV:Body> <add>
 <result xsi:type='xsd:int'>1033276
</result> </add>
</SOAP-ENV:Body>
```

(c) P3's Response Message

header attributes

```
id            ="b5f9bd2d..942b3bd88"
s:type        ="response"
s:issuedAt    ="2010-06-14T09:04:00"
deadline      ="2010-06-15T09:00:00"
publisher     ="P4"
destination   ="alicemobile"
s:service     ="S3"
s:capability  ="sum,add"
s:requestID   ="b89f6db2..38ca870d"
s:requestAt   ="2010-06-14T09:00:00"
s:resPolicy   ="first"
s:healing     ="aggressive"
s:doubt       ="2"
```

payload

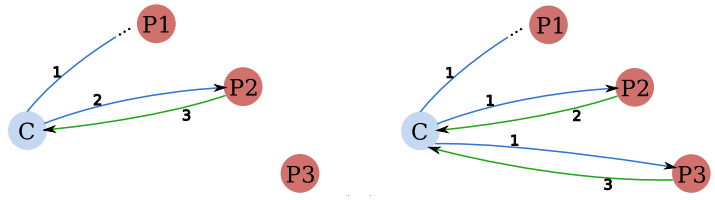
```
<SOAP-ENV:Body> <add>
<result xsi:type='xsd:float'>1033276
</result> </add>
</SOAP-ENV:Body>
```

(d) P4's Response Message

Figure 6.5: Example response messages

6.3.3 Response management policy

The default invocation in DiSWAN uses content-based request dissemination that widens the invocation addressing range to a multiplicity of providers, which I call group invocation. Figure 6.6 compares a traditional destination-based invocation behavior to DiSWAN's content-based group invocation:



(a) Destination-based client orchestration.

(b) Content-based group invocation.

Figure 6.6: Destination Vs content-based invocation

(a) Client orchestration relies on sequential invocations. After the failure of the first invocation attempt (due deadline), the client issued the second request to another provider. Therefore, the client gets satisfied after: $t_a = \texttt{deadline} + RTT(C, P2)$.

(b) With group invocation, the client targets all possible providers with a single request. Therefore, the client gets satisfied after: $t_b = RTT(C, P2)$.

Addressing a business service instead of a unique provider increases the reactivity of the system but also possibly generates multiple responses for a single invocation request. The service platform enables the client to manage this multiplicity using two policies: *multiple-response* and *first-response*. The response management policy is chosen by the client and specified in each invocation request using the `s:resPolicy` attribute (either `"multiple"` or `"first"`).

Multiple-response policy.

Using a *multiple-response* policy lets all responses to a client's request be delivered to this client. Indeed, there are situations in which a client could see it as an advantage. Consider for example a client using a service designed for translating an English sentence into French. Multiple responses can be useful to an end-user who can choose the best answer. Similarly, it could be useful that a call to a temperature service in a sensor network let multiple responses reach the client which would average the received values.

First-response policy.

On the other hand, in the case where only one response is useful, the client applies a *first-response* policy, in which only the first received response is interesting to the client and the following ones are discarded. A train schedule information service for example allowing requests such as "at what time are the departures to Paris between 12:00 and 18:00" is normally invoked with a *first-response* policy because any extra response received after the first one is obviously redundant.

6.3.4 Network healing

After the client gets satisfied, the invocation leftovers will continue to propagate until their deadline is reached, consuming network bandwidth and storage capacity in the relay nodes. Moreover, benefiting from the fact that several providers are allowed to answer a single invocation request incurs extra network cost, specially in case only one response is expected by the client. Network healing is the elimination of the invocation leftovers, including redundant responses and the request message itself.

Network healing is only applied in case the client requests a first-response policy. When the client requests a multiple-response policy, the client expects to receive multiple responses until the deadline defined in the request. When the deadline is reached, the invocation completes and the request and response messages become obsolete and are naturally deleted by network nodes.

In disconnected and dynamic mobile environments, consensus is not applicable because the set of providers is not known a priori and the absence of full connectivity of the network precludes learning its composition in a reasonable time. Consensus protocols between providers can guarantee the uniqueness of an invocation (e.g. unique transaction) and generally reduce network overhead. A consensus protocol normally assigns one delegated responder at a given time. My objective is more realistic and therefore less ambitious: it will not guarantee that only one response reaches the client node but rather tries to cancel the propagation of redundant messages.

A low-level network healing ability is provided by the DoDWAN communication layer via the `cancel(id, deadline)` method (see Section 4.3.4 in Chapter 4). A service agent can cancel a message using the message's unique id. The cancel order propagates to network nodes through the low-level periodic catalog announcements until the deadline. In other words, the cancel order travels from one network node to another using the catalog announcements. When a node receives the cancel order, it deletes the message from its cache, stops relaying this message, and appends the cancel order to its own catalog announcements.

A node can call the cancel method only if it knows the message's id. Since an invocation can produce many response messages from different providers, then a node wanting to initiate the healing of the network does not necessarily know all the response messages in order to be able to cancel them. Hence a node wanting to initiate the healing creates a healing message in addition to canceling the known messages. The healing message helps other nodes cancel corresponding messages, these corresponding messages are not known to the initiating node.

DiSWAN enables the client to request one of two types of healing: safe healing and aggressive healing.

- The client requests the safe healing type by putting the attribute `s:healing="safe"` in the invocation request. This type of healing makes use of the low-level cancel method, it also enables a reactive competition technique between providers during the invocation, in addition to creating a client-initiated healing message after satisfaction.
- The client requests the aggressive healing type by putting the attribute `s:healing ="aggressive"` in the invocation request. This type of healing makes use of the

low-level cancel method, it also enables a reactive competition technique between providers during the invocation, in addition to creating provider-initiated healing messages even before the satisfaction of the client.

In the following, I present the reactive competition technique, I also present the safe and aggressive healing types.

6.3.4.1 Reactive competition

Using either the safe or the aggressive healing types, a reactive competition technique is always applied. Reactive competition is applied only at provider nodes, it does not involve creating healing messages. A provider subscribes for messages holding responses to requests it could himself respond to. When a provider P1 receives such a response issued by another provider P2, then:

- If P1 has not yet himself answered the request: P1 compares the `s:doubt` attribute of P2's response with the one it can provide itself. If P2's response has a lower or equal doubt (better or equal reliability), then it gives up attempting to answer the request and rather starts to relay P2's response. Otherwise, it answers the request and cancels P2's response.
- If P1 had already answered the request: P1 also compares the `s:doubt` attribute of P2's response with its proper response. P1 keeps the most reliable answer by canceling the response with a higher doubt. In case both answers have equal doubts, then it compares the issued date of both responses and cancels the more recent one.

Reactive competition is always profitable in terms of reduction of the number of transmitted messages (no extra messages are added) and it does not delay the arrival of the first response at the client.

6.3.4.2 Safe healing

The safe healing type (`s:healing="safe"`) applies the basic reactive competition during the invocation. When a client gets satisfied by receiving its first response, it cancels its request message and cancels the received response message (both messages are known to the client). In addition to canceling the known messages, the client creates and publishes a control message called healing message in order to try to eliminate from the network other messages that are not known to the client. This client-initiated healing message disseminates in the network. When a node λ receives this healing message, it cancels response messages that correspond to the denoted invocation. Figure 6.7 shows an example client-initiated healing message.

The healing message contains only a header and no payload. The `s:type= "healing"` attribute defines the message as a healing message. The `deadline` attribute's value is the same as the invocation deadline. The `s:requestID` attribute contains the id of the request message. A large number of nodes, if not all, are supposed to subscribe for control messages of this type in order to ensure their rapid dissemination.

In effect, when a node λ receives a healing message, it creates a healing predicate. Figure 6.8 shows a healing predicate corresponding to a client-initiated healing message.

header attributes	
id	="e3185af50..b7868f25"
s:type	="healing"
deadline	="2010-06-15T09:00:00"
s:initiated	="client"
s:requestID	="b89f6db2..38ca870d"
empty payload	

Figure 6.7: Client-initiated healing message example

$h_C =$ `s:type="response"` $\wedge$ `s:requestID="b89f6db2b9515b88cc93b7ee38ca870d"`

Figure 6.8: Healing predicate corresponding to the client's healing message of Figure 6.7

Node λ creates a filter using the predicate via a `filter(predicate, deadline)` method to filter local messages. The filter interface is similar to the matching of the subscribe interface, but only applies to local messages present in the cache of the node, it does not involve asking other nodes for messages. The created filter stays active until the deadline of the invocation. The filter created with the predicate of Figure 6.8 matches response messages that have the specified `s:requestID` attribute. Filtered messages are canceled by node λ.

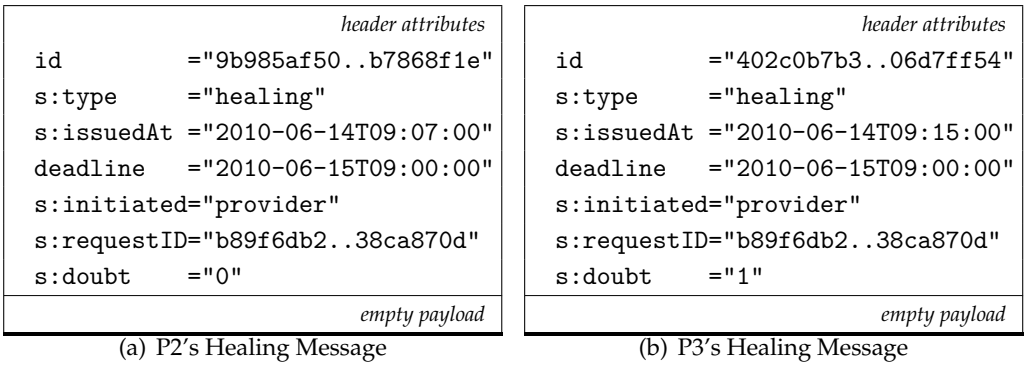
As extra messages are transmitted, this tends to increase the network load, yet healing messages have a very small size so the elimination of pending responses –whose size can be orders of magnitude larger– compensate in most cases for this augmentation (this is shown in the simulations of Chapter 7).

6.3.4.3 Aggressive healing

The aggressive healing type (`s:healing="aggressive"`) applies the basic reactive competition during the invocation. It also consists in starting the dissemination of healing messages by the providers before a response is known to have reached the client. A provider P answering a request creates and starts disseminating a healing message (`s:type= "healing"`). In other words, the provider publishes a response message and also publishes a corresponding healing message. Figure 6.9 shows example healing messages.

A healing message from P is a command to other nodes to cancel all the instances of the specified request as well as all the corresponding responses of lower quality than the response issued by P. A response message m is considered as of lower quality in comparison to another response message n, if m has a doubt higher than the doubt of n, or if m is issued after n in case the doubt is equal for both.

In effect, when a node λ receives a healing message, it creates a corresponding healing predicate. Node λ creates a filter using this predicate via the `filter(predicate, deadline)` method to filter local messages. The filter only applies to local messages present in the cache of the node, it does not involve asking other nodes for messages. The created filter stays active until the deadline of the invocation. Filtered messages are



(a) P2's Healing Message (b) P3's Healing Message

Figure 6.9: Provider-initiated healing message examples

cancelled by the node.

For example, when node λ receives P2's healing message, λ creates the corresponding healing predicate h_{P2} (see Figure 6.10).

$$h_{\mathbf{P2}} = (\texttt{s:type="response"} \vee \texttt{s:type="healing"}) \wedge (\texttt{s:requestID="b89f6db2b9515b88cc93b7ee38ca870d"}) \wedge (\texttt{s:doubt>"0"} \vee (\texttt{s:doubt="0"} \wedge \texttt{s:issuedAt>"2010-06-14T09:07:00"}))$$

Figure 6.10: Healing predicate corresponding to P2's healing message

The filter created with h_{P2} filters response messages answering to the same request id, and having a higher doubt factor than P2, or having an equal doubt factor but were issued after P2's response. Filtered messages are cancelled. If for example P4's response message is received by λ, it gets filtered according to h_{P2} because it responds to the same request as P2 and has a higher doubt than P2, then λ cancels P4's response message using: `cancel("b5f9bd2d..942b3bd88", "2010-06-15T09:00:00")`.

h_{P2} also filters healing messages. In case a network node receives two healing messages for the same request, it will only consider the one having a smaller doubt factor or with an equal doubt factor but issued beforehand. The other healing message is cancelled.

In addition, when a node λ receives a healing message, it cancels the corresponding request message only if the healing message presents a doubt of `"0"`. In other words, λ can cancel the request because it is now sure that there exists a reliable response in the network. For example, when λ receives P2's healing message then it cancels the request, but when it receives P3's healing message it will not cancel the request. The node cancels the request using:
`cancel("b89f6db2..38ca870d", "2010-06-15T09:00:00")`

The client node gets satisfied after receiving a response (e.g. P2's response), so it cancels this response message from the network using:
`cancel("69ae4c15d..45fb1c1a", "2010-06-15T09:00:00")`.

A healing message is useful only if it is relayed by more nodes than the providers of

the concerned service, therefore this healing message should be relayed by almost all the nodes of the network. The objective of a provider-initiated healing message is to enable at simple nodes, a behavior similar to the reactive competition behavior at provider nodes, and consequently this eliminates more redundant messages. However, in unfavorable network conditions, it may happen that a healing message provokes the cancellation of a response that could otherwise have been the first one to arrive at the client. Thus, this healing type may theoretically delay the arrival of the response to the client. Experiments I conducted tend to show that this delay is negligible whereas the gain in redundant message elimination is significant (see Section 7.5.2 in the next chapter).

6.3.5 Discussion

6.3.5.1 Client-Provider binding

A binding represents the reference exchange between a client and a provider in order for invocation interactions to work. Traditional client-server computing models have a strong binding between the two entities: a client –as soon as it is created– is bound to a server for all future interactions. The service-oriented computing came into the picture with its late binding.

Late binding.
The idea of late binding characterizes the service computing model, in that the client does not need to be bound to a provider before actually invoking it. As seen in Figure 6.11, from the point in time t_0 when a client discovered a service provider P1 to the point t_1 when the client started an invocation, the client was not bound to any provider.

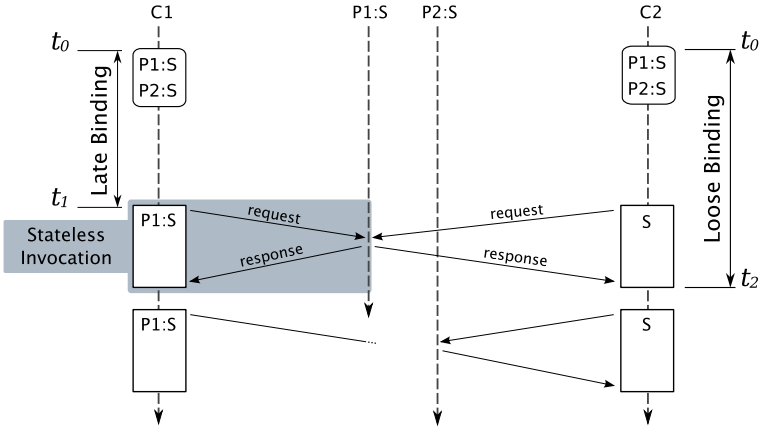


Figure 6.11: Stateless invocations, late binding compared to loose binding

Loose binding.
In an effort to push this late binding even more, DiSWAN proposes loose binding, it allows the client to invoke a service instance without binding to the actual provider hosting this service. The client publishes a request message to a service group, the request flows without forcing an explicit provider's destination. As seen in Figure 6.11, from the

point in time t_0 when a client discovered a service provider P1 to the point t_2 when the client received an invocation response, the client was not bound to any provider. This is made possible by appending a reduced version of the service descriptor (`s:service` and `s:capability`) to the header of the request in order to let other providers receive the request and decide whether or not they can answer. Hence a client formulates its request not only with a SOAP part according to a discovered service, but also with a description of the invoked service in the header of the request message.

Symmetrically, a provider does not receive only a request in which its name figures in the request's destination attribute. Instead, a provider subscribes so that it receives all the requests containing a header matching one of the hosted services.

6.3.5.2 Loose coupling benefits

My focus in the design of the service platform is to enhance loose coupling between clients and providers. The design helps decrease the number of needed interactions, it provides less dependency, and helps decrease the latency of the request-response interaction cycle.

The discovery behavior decreases the interactions between the client and provider entities to a minimum. The discovery protocol is an always active process, where the client continuously discovers interesting service providers. When this client needs to use a service, it generates an invocation request.

Furthermore, with the above detailed invocation solution: it becomes possible for a client to invoke a desired business service without special regard to an actual hosting provider and where several providers hosting the same business service deal with this type of invocation. Therefore the content-based invocation enables less dependency between a client and service providers.

The invocation solution enables all compatible providers to answer the client. The response of the closest provider gets received by the client, therefore decreasing the round trip time in comparison to that with far away providers.

On the downside, using capability matching induces more doubt about the reliability of the answering provider. In other words, if the `add` capability of P3 matches the `add` capability of P1 syntactically but not semantically, then P3's response would not be valid for the invoking client any more. Alias capabilities are even harder to match, the `sum` capability must match the `add` capability semantically, which induces even more doubt about the reliability of P4's response.

Figure 6.12 illustrates how higher loose coupling provides higher doubt and therefore less reliability. If the invocation request is only addressed to a specific provider destination then the invocation is the least loose coupled. If the request could also be addressed to the service hosted by many providers then the invocation is more loose coupled. If the request can address all the possible compatible providers as enabled by the default invocation behavior (capability and alias capability matching), then loose coupling increases but with decreased reliability.

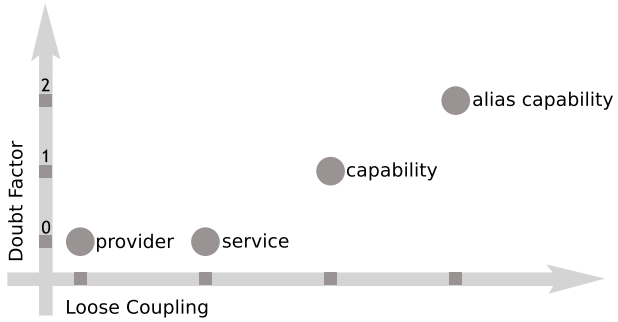


Figure 6.12: Pushing loose coupling to the limits may produce a higher doubt about the answering provider's reliability.

6.4 Invocation restrictions

In order to make it possible for a client to invoke a desired business service without special regard to an actual hosting provider, DiSWAN enables using content-based invocations. A content-based invocation addresses a wide range of possible providers. However, DiSWAN also lets this client application orchestrate invocations by selecting then requesting a specific provider or a specific service. This behavior is considered as a restriction to the content-based default invocation solution proposed above, by rolling back to a more traditional service invocation behavior. Implementing these restrictions is rather straightforward, by making modifications to the request message.

header attributes

```
id          ="b89f6db2..38ca870d"
s:type      ="request"
s:issuedAt  ="2010-06-14T09:00:00"
deadline    ="2010-06-15T09:00:00"
publisher   ="alicemobile"
destination="P1"
s:service   ="S1"
```

payload

```
<SOAP-ENV:Body>
 <add>
<number1 xsi:type='xsd:int'>965...
<number2 xsi:type='xsd:int'>676...
 </add>
</SOAP-ENV:Body>
```

(a) Service Restricted Request

header attributes

```
id          ="b89f6db2..38ca870d"
s:type      ="request"
s:issuedAt  ="2010-06-14T09:00:00"
deadline    ="2010-06-15T09:00:00"
publisher   ="alicemobile"
destination="P1"
```

payload

```
<SOAP-ENV:Body>
 <S1>
 <add>
<number1 xsi:type='xsd:int'>965...
<number2 xsi:type='xsd:int'>676...
 </add>
 </S1>
</SOAP-ENV:Body>
```

(b) Provider Restricted Request

Figure 6.13: Request messages for restricted invocations

- A client restricts the invocation to a specific service by moving the `s:capability` attribute from the request's header and placing it directly in the SOAP payload (see Figure 6.13(a)).

- A client restricts invocation to a specific provider by moving the `s:service` attribute from the request's header and placing it directly in the SOAP payload (see Figure 6.13(b)).

For example, there exists situations in which a specific provider is expected to be addressed when transferring the invocation request. This can occur because a provider is known to be the only one to ensure a certain quality of service or because of the very nature of the service. In some situations the client prefers to restrict the invocation to a specific provider according to contextual availability in time and space.

Because of the client orchestration, network healing is not the same as described above. After the client gets satisfied, it simply cancels the request and response messages in order to heal the network.

Figure 6.14 illustrates restricted invocation scenarios (a) and (b). Scenarios (c) and (d) illustrate the default invocation (Section 6.3).

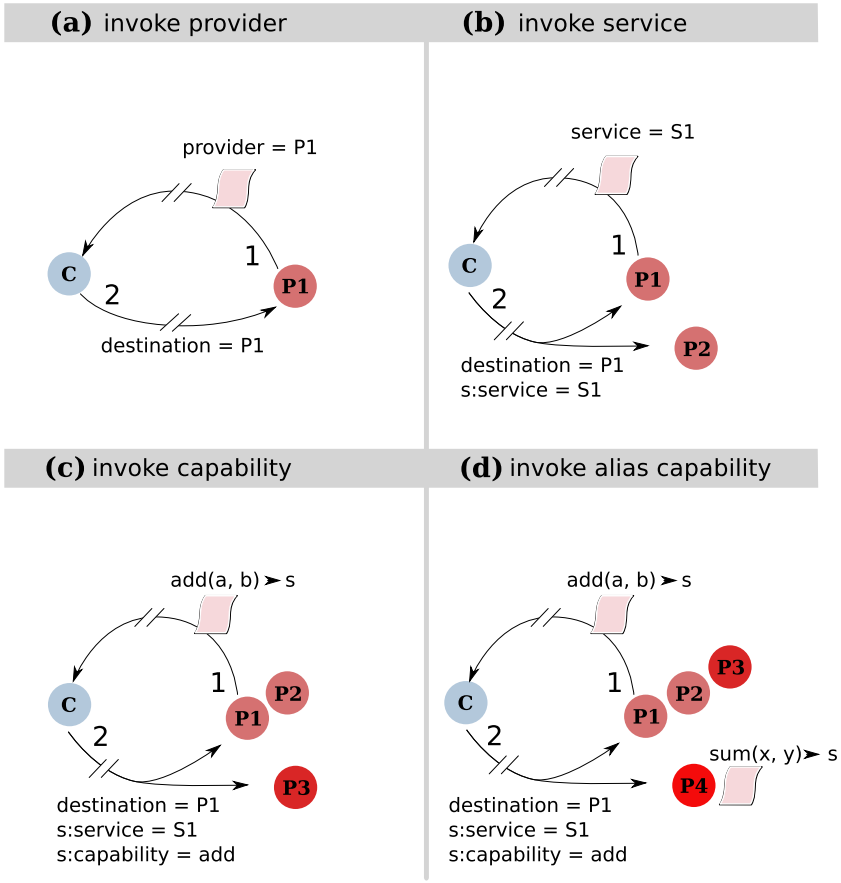


Figure 6.14: Multi-level addressing in a request enables a wider impact range

(a) After discovering the provider P1, the client's invocation request is published to disseminate in the network. Using the `destination` attribute, the request message can be delivered to the exact provider P1.

(b) The client appends the `s:service` attribute to its request's header. Therefore, the request message can be delivered to providers P1 and P2 (both hosting the service S1). Service level matching removes the client's dependence on a specific provider, thus enabling this client to address many providers of the same service.

(c) The client appends the `s:capability` attribute to its request's header. Therefore, the request message can be delivered to providers P1, P2 and P3 (all offering the same capability). Capability matching provides even more independence by expanding the range of possibly addressed providers, where the requested capability is exposed by different services.

(d) As an extension of scenario (c), the request message can be delivered to providers P1, P2, P3 and P4, using alias capability matching at the P4 provider.

6.5 Blind invocation: Bypassing discovery

In all the previously described invocation solutions, the invocation process follows a discovery phase where a client gets to know a provider's reference or just its functional resources in order to create a well formed request. In case the client had not received any interesting service descriptors when an invocation is needed, the client sends a blind invocation request (see example in Figure 6.15).

header attributes

```
id           ="b89f6db2b9515b88cc93b7ee38ca870d"
s:type       ="request"
s:issuedAt   ="2010-06-14T09:00:00"
deadline     ="2010-06-15T09:00:00"
publisher    ="alicemobile"
s:capability ="add|sum"
```

payload

```
<SOAP-ENV:Body>
 <operation>
    <input xsi:type='xsd:int'>965654</input>
     <input xsi:type='xsd:int'>67622</input>
 </operation>
</SOAP-ENV:Body>
```

Figure 6.15: Blind request message

Since there is no discovered service providers, a blind request's header cannot have neither `destination` nor `s:service` attributes. A blind header focuses on the functional invocation `s:capability` attribute. The `s:capability` attributes could be the same attribute found in the client's original discovery predicate . With this behavior, instead of waiting to discover service descriptors, a blind request completely removes the destination and service attributes. In other words, the client side matching for service discovery previously used in the discovery protocol is replaced with provider side matching, consequently making an invocation without a client's discovery process.

Since there is no discovered functional interface, a blind request does not have an accurate SOAP invocation syntax. Instead, the client uses generic operation and input SOAP elements in the request's payload (see Figure 6.15).

On the other side, in order to receive blind requests, a provider subscribes using a subscription predicate of the following format:

$$p_{\mathrm{P}} = \texttt{s:type="request"} \wedge \texttt{s:capability="add"} \tag{6.7}$$

When a service provider receives a blind request, it retrieves the input values and creates a response message with a `s:doubt="2"` attribute.

Blind invocation enhances the service provision process by completely bypassing the whole discovery phase. Blind requests relying on capability matching and on the providers intelligence, reduces needed interactions, thus boosting loose coupling. This invocation scenario provides the most loose coupling that our service platform can afford to do. Full semantic invocation scenarios using ontologies are the obvious next step for totally loose coupled service provision. The proposed platform does not implement such semantic invocation.

6.6 Client and provider states

During discovery and invocation, clients and providers go through many states. As a recapitulation of the service provision process, Figure 6.16 shows the state diagrams of both clients and providers (healing interactions are not shown for the sake of simplicity).

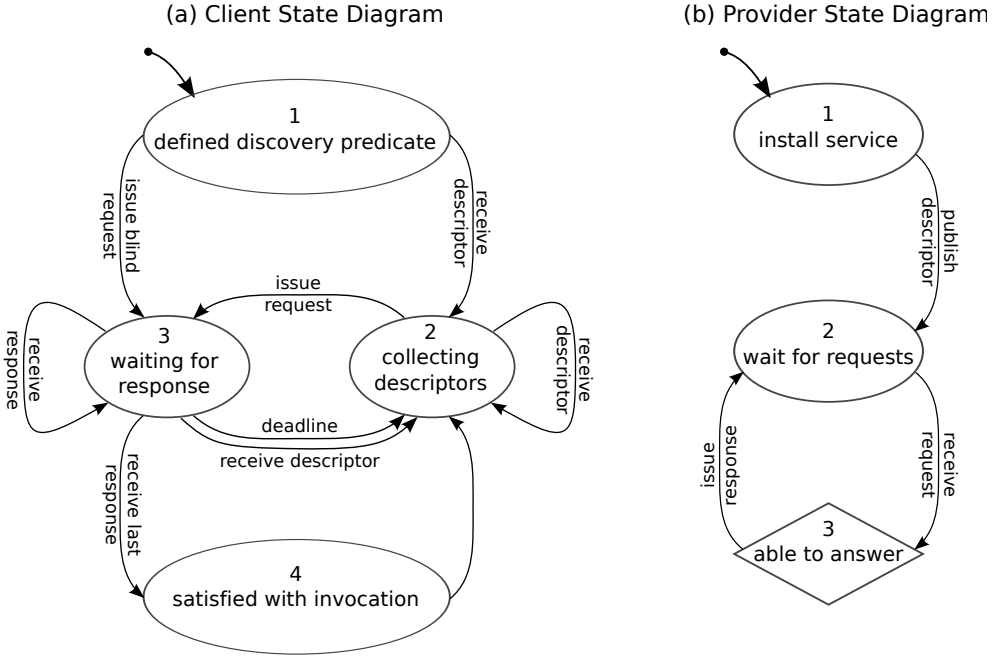


Figure 6.16: Client and provider states

A client defines a discovery predicate, and has the option of either waiting for service descriptors in order to issue normal requests or urgently issue blind requests. After

publishing its request, a client waits for a response so that it becomes satisfied with invocation. A provider installs services, publishes its descriptor, subscribes and waits for requests, received requests are processed and responses are published back to the requesters.

6.7 Remote invocations of stateful services

By definition, stateful services are services preserving state information, where an invocation may depend on the results of previous invocations. A stateful service is capable of keeping track of a client through the use of a persistent session accessible only by this client. A service saves the conversational state data of its corresponding client. The state data is stored in the memory of the service. The client associates itself with a single service where its session resides. A session has a predefined inactivity deadline corresponding to the maximum time allowed between two client invocations. In other words, if the client does not invoke the service within the deadline time after its last invocation, then the session will be lost.

Stateful services can be easily implemented where temporaneous communications are available, for example if the client and provider are in proximity or if they reside in connected networks. Session loss occurs in case the provider becomes unreachable for a time that exceeds the session deadline. In network environments where communications are not guaranteed like in disconnected MANETs, session loss becomes more frequent. To enable stateful service invocations in disconnected MANETs, session management mechanisms must be implemented. Session management should provide session recovery on another provider in case the session is lost at the original provider.

When services have to maintain state data in memory (i.e. stateful session) between invocations, the client becomes strongly bound to the provider. State messaging [109] presents a solution, it delegates the storage of state data to the exchanged messages instead of saving the state data in memory. The state data is added to the request and response messages in order to keep the other party up-to-date. When a client issues a request to a service provider, the provider creates the necessary data structures to maintain the state and appends this state data to the response. The client processes the response and updates the state data that is appended to the next request message.

The solution I propose builds on the same state messaging mechanism. In case an originally intended provider would become unreachable, the client holds the needed state information enabling another compatible provider to resume.

Stateful services presume that there would be sequential request-response cycles. The first client request opens a new session at the provider, the first response returns the session identifier to the client, so that the latter could work with its session later on. The first request-response cycle is considered as a stateless invocation, so that the default invocation solution previously presented still apply. Once the client receives the first response from a provider, then all consecutive requests will be destined to this provider. A session context holding up-to-date state information is returned with every response, so that when the sessions is lost, another provider can resume the session from where it ended.

Let's take the example application of a mortgage calculator. The service calculates monthly payments based on the global amount, the number of years, and the interest rate. Figure 6.17(a) shows the first client request message: attribute `s:session="0"` asks the provider to open a session, and the payload contains the mortgage capability's SOAP inputs. Figure 6.17(b) shows the the service's response: the `s:session` attribute holds the session's identifier for future reference, the payload contains the state data as well as the SOAP response. Figure 6.17(c) shows the second client request: containing the session reference, the state data, and invoking the `extra` capability to check how an extra monthly payment reduces the number of years. By the time the client sends its second request, suppose provider P2 has become out of reach. Since the second request has the session context, it would not be hard for provider P1 to pick up from where P2 left off. Figure 6.17(d) shows P1's response: containing the same session identifier to show the client that his session was recovered, and the updated state data, and the SOAP response.

header attributes

```
id           ="b89f6db2..38ca870d"
s:type       ="request"
s:session    ="0"
s:issuedAt   ="2010-06-14T09:00:00"
deadline     ="2010-06-14T09:20:00"
source       ="alicemobile"
destination  ="P1"
s:service    ="S1"
s:capability ="mortgage"
```

```
<SOAP-ENV:Body>
<mortgage>
 <amount xsi:type='xsd:int'>170000</amount>
 <years xsi:type='xsd:int'>20</years>
 <interest xsi:type='xsd:int'>7</interest>
</mortgage>
</SOAP-ENV:Body>
```

(a) First Request Message

header attributes

```
id           ="69ae4c15d..45fb1c1a"
s:type       ="response"
s:session    ="UFG7656ZDT"
s:issuedAt   ="2010-06-14T09:02:00"
deadline     ="2010-06-14T09:20:00"
source       ="P2"
destination  ="alicemobile"
s:service    ="S1"
s:capability ="mortgage"
s:requestID  ="b89f6db2..38ca870d"
s:requestAt  ="2010-06-14T09:00:00"
s:doubt      ="0"
```

```
<state>
 <amount xsi:type='xsd:int'>170000</amount>
 <years xsi:type='xsd:int'>20</years>
 <interest xsi:type='xsd:int'>7</interest>
</state>
<SOAP-ENV:Body>
 <mortgage>
  <monthly xsi:type='xsd:int'>1318</monthly>
 </mortgage> </SOAP-ENV:Body>
```

(b) P2's Response to (a)

header attributes

```
id           ="19c6d7e3d..a6ff6085"
s:type       ="request"
s:session    ="UFG7656ZDT"
s:issuedAt   ="2010-06-14T09:10:00"
deadline     ="2010-06-14T09:30:00"
source       ="alicemobile"
destination  ="P2"
s:service    ="S1"
s:capability ="extra"
```

```
<state>
 <amount xsi:type='xsd:int'>170000</amount>
 <years xsi:type='xsd:int'>20</years>
 <interest xsi:type='xsd:int'>7</interest>
</state>
<SOAP-ENV:Body>
 <extra>
   <monthly xsi:type='xsd:int'>200</monthly>
 </extra> </SOAP-ENV:Body>
```

(c) Second Request Message

header attributes

```
id           ="d879d02c2..8b662e5"
s:type       ="response"
s:session    ="UFG7656ZDT"
s:issuedAt   ="2010-06-14T09:13:00"
deadline     ="2010-06-14T09:30:00"
source       ="P1"
destination  ="alicemobile"
s:service    ="S1"
s:capability ="extra"
s:requestID  ="19c6d7e3d..a6ff6085"
s:requestAt  ="2010-06-14T09:10:00"
s:doubt      ="0"
```

```
<state>
 <amount xsi:type='xsd:int'>170000</amount>
 <years xsi:type='xsd:int'>15</years>
 <interest xsi:type='xsd:int'>7</interest>
</state>
<SOAP-ENV:Body>
 <extra>
  <years xsi:type='xsd:int'>15</years>
 </extra> </SOAP-ENV:Body>
```

(d) P1's Response to (c)

Figure 6.17: Stateful sequential invocation messages

6.8 Public invocations

All previous invocation examples of this chapter, are private scope invocations. A requesting client is the final destination of a provider's response. In other words, a response message is useless to some other client node, even though this other client node might be willing to relay the message.

Public scope.
An invocation is of public scope if the provider's response may interest many possible clients. This type of behavior relates more to information sharing applications, like a weather information service for example. Figure 6.18 shows a public request-response invocation example of a weather service.

header attributes

```
id            ="7a3430ad..58ca8d5a"
s:type        ="request"
s:issuedAt    ="2010-06-10T09:00:00"
deadline      ="2010-06-10T09:20:00"
source        ="alicemobile|public"
destination   ="P1"
s:service     ="S1"
s:capability  ="weather"
```

payload

```
<SOAP-ENV:Body>
 <weather>
  <city xsi:type='xsd:string'>vannes</city>
 </weather>
</SOAP-ENV:Body>
```

(a) Public Request Message

header attributes

```
id            ="9166c320..df74f9b34"
s:type        ="response"
s:issuedAt    ="2010-06-10T09:02:00"
deadline      ="2010-06-11T00:00:00"
source        ="P1"
destination   ="alicemobile|public"
s:service     ="S1"
s:capability  ="weather"
s:requestID   ="7a3430ad..58ca8d5a"
s:requestAt   ="2010-06-14T09:00:00"
s:doubt       ="0"      />
```

binary payload

```
<requestSOAP>
 <weather>
  <city xsi:type='xsd:string'>vannes</city>
                    </weather> </requestSOAP>
<SOAP-ENV:Body>
 <weather>
  <temperature xsi:type='xsd:int'>16
                                </temperature>
 <html>...
```


```
      ...</html> </weather> </SOAP-ENV:Body>
```

(b) Public Response Message

Figure 6.18: Public invocation messages

The client issues its request with a `source="alicemobile|public"` attribute, with

the SOAP input in the payload. The provider answers with a message containing the `destination="alicemobile|public attribute"`, the SOAP response appended to the SOAP request in the payload. For private invocations, the deadline needed by a client is supposed to be the same for the request and the response messages. In other words, there is no need for these messages to stay alive and cached in the network beyond the requesting client's deadline. In the case of a public scope invocation, the response may still be valid after the requesting client's deadline. Thus this response is given a proper deadline in order for other clients to benefit from it. In the above weather example, the returned weather information will stay valid for the rest of the day, therefore the response's deadline is set accordingly.

6.9 Perspectives

6.9.1 Semantic invocation

Future mobile devices will have an increase in computing and storage power, that will enable them to handle complex ontology languages and semantic systems. That will eventually allow semantic only invocations, where no a priori discovery is needed. For example, a client sends an English written request that will semantically be directed and handled to the proper service that will semantically process and respond back to the client. Even though our blind request can somehow achieve this same goal, but it is based on semantic keyword comparison and filtering and does not resort to any ontologies. My main reason not to use ontologies is that the ontology language has to be known by all the entities of the service platform, this situation can be divided in two points:

- Reduced size specific domain ontologies that must be distributed to all service entities. Problems occur if this ontology is updated on the service side and not on the client side for example.

- Full blown ontology languages including specific and large domain ontologies and spoken language ontologies. Actual mobile devices are not capable of handling the large storage space nor the computational power needed for such semantic systems.

6.9.2 Complex request

Since a request-response cycle's completion time can be unpredictably long, it is helpful to group sequential requests into a one time complex request. Of course this is not obvious for most stateful applications since the client usually issues a stateful request after checking the result of a previous response. But a simple use case of a complex request would be to use the same capability many times with different input variations (see example in Figure 6.19).

header attributes

```
id            ="c90b0fde6..e2852b47"
s:type        ="request"
s:session     ="0"
s:issuedAt    ="2010-06-14T09:00:00"
deadline      ="2010-06-14T09:20:00"
source        ="alicemobile"
destination   ="P1"
s:service     ="S1"
s:capability  ="mortgage"
```

binary payload

```
<SOAP-ENV:Body>
<mortgage>
 <amount xsi:type='xsd:int'>170000</amount>
 <years xsi:type='xsd:int'>20</years>
 <interest xsi:type='xsd:int'>7</interest>
</mortgage>
</SOAP-ENV:Body>

<SOAP-ENV:Body>
<mortgage>
 <amount xsi:type='xsd:int'>170000</amount>
 <years xsi:type='xsd:int'>15</years>
 <interest xsi:type='xsd:int'>7</interest>
</mortgage>
</SOAP-ENV:Body>
```

Figure 6.19: A complex request example

6.10 Summary

This chapter described the full invocation solutions of the service platform. Invocations must be tolerant to communication delays due to disconnected network environments. The chapter presented the service provider's kinds found in mobile environments, where there might be more than one providers offering the same business service. The chapter presented remote invocation solutions, detailing how a client can issue a content-based invocation request to all providers of a business service, and how these providers respond back, and how network healing techniques eliminate unneeded messages. It also presented how a client can restrict the invocation to specific providers. In case of a discovery failure, a client can even send a blind invocation request. The chapter also presented stateful remote invocation solutions via session management. Public invocations solutions were presented, for cases where a response can benefit more than one requesting client. And it presented a future perspectives. The next chapter presents the implementation of the service platform as a middleware, and shows evaluations using simulated network scenarios.

7 Implementation and Evaluation

Contents

7.1 Introduction 83
7.2 The service platform's middleware 83
7.3 Simulation environment and evaluation metrics 86
7.4 Discovery evaluation 87
7.5 Invocation evaluation 89
7.5.1 Effects of invoking multiple providers 89
7.5.2 Network healing 91
7.5.3 Session recovery for stateful invocations 93
7.6 Conclusion 94

7.1 Introduction

This chapter details the middleware implementation of the service platform described in the previous chapters. Section 7.2 presents the middleware's layout. It consists of a service layer where service agents reside, they access the communication layer via a publish/subscribe interface. Using this implementation, I conducted simulations in as realistic as possible conditions in order to evaluate the service platform (Section 7.3). The main objective is to assess the performance of service discovery (Section 7.4) and invocation (Section 7.5) in terms of response time and network load. Section 7.6 concludes the chapter.

7.2 The service platform's middleware

The service platform is structured in two layers. Figure 7.1 gives an overview of its architecture. The first layer (DiSWAN) is a high-level service layer that is in charge of all service-oriented processing, enabling discovery and invocation interactions between clients and providers. The second layer (DoDWAN) is a communication layer taking care of all data level exchanges, resulting in a communication system fully adapted to disconnected network environments.

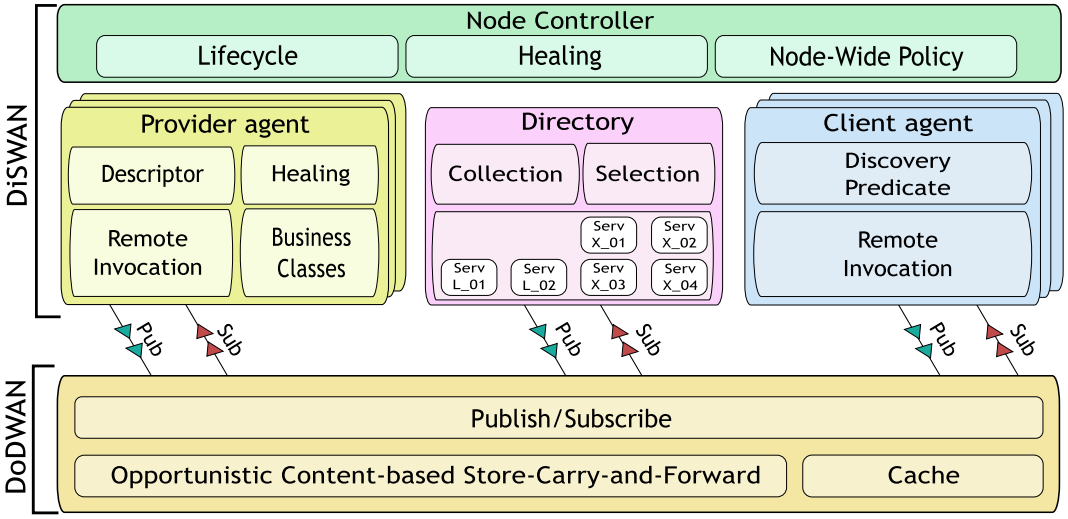


Figure 7.1: Architecture of the service platform

From the service layer's point of view, a message is sent from a source node to the network using a message publication. The message is disseminated in the network according to its content; it is received at a node that has previously provided a predicate matching this message, via a subscription operation. Even though the architecture separates the service and communication layers, their are interconnected via the publish/-subscribe interface. Messages created by the service layer are directly exploitable by the communication layer, and informations related to the communication layer are accessible to the service layer.

One service platform is operational on each network node. It enables a node to act as a provider and/or a client. The service platform can host several provider agents and several client agents. The service delivery is carried out in two phases of client-provider interactions. The first phase is the service discovery during which a provider should advertise its service in the network for potential clients to discover. The second phase is the service invocation during which a client invokes providers. The communication protocol was described in Chapter 4, and the discovery and invocation protocols were described in Chapters 5 and 6.

- Each provider agent exposes an installed business service. When the user installs a service, a provider agent is created. The provider agent is responsible for creating the descriptor message. It puts the description of functional and non-functional service properties as well as the node's context properties in the descriptor message's payload. It fills the descriptor message's header according to the description information. It advertises the descriptor by publishing it to the network. The provider agent also acts as a container for installed business classes. It subscribes to receive request messages, executes the service invocation, and then publishes response messages to the invoking clients. The provider agent also performs the reactive competition healing technique and creates provider-initiated healing messages.

- Each client agent needs to consume an exposed service corresponding to a defined

discovery predicate. When the user defines a set of needs using a discovery predicate, a client agent is created. The client agent subscribes for interesting services using the predicate. Received descriptors are collected in the directory. When an invocation is needed, the client agent selects a descriptor from the directory. It creates and publishes request messages, and subscribes to receive response messages. According to the user's needs, the client agent defines the response policy, which is either multiple-response or first-response. It also defines the type of healing, which is either safe or aggressive. In case the healing is safe, the client agent creates client-initiated healing messages after satisfaction.

- The directory module acts as a storage of discovered service descriptors. It collects the descriptors that interest the client agents. It provides a selection interface used by a client agent to get contextually appropriate descriptors. It also can notify a client agent of the presence of a contextually appropriate descriptor in order to opportunistically start an invocation.

- The node controller manages the lifecycle of the provider agent and client agent modules. The controller applies node-wide preferences. It takes the human user's general interests, it subscribes using these interests for matching messages, which helps the node participate in relaying messages that are not necessarily used by the provider and client agents. This altruist behavior is generally recommended, specially in highly disrupted network environments. The controller also subscribes for healing messages in order to make the node participate in network healing. It cancels cached messages that correspond to a received healing message.

- The DoDWAN communication layer implements a publish/subscribe interface over a store-carry-and-forward module in order to enable content-based message dissemination. It holds the node's cache where all the messages published by the upper layer and received from the network are stored. The cache is responsible for deleting obsolete messages (due deadline).

A proof of concept prototype of DiSWAN is implemented in Java (5500 lines of code). The `wsdl4j.jar` [1] library is used for manipulating WSDL descriptions, and the `saaj.jar` [2] library is used for manipulating SOAP invocation requests and responses. The lower layer uses a Java implementation of the DoDWAN [3] protocol.

In order to assess the performance of the service platform, the middleware is interfaced with the MADHOC network simulator [110]. MADHOC provides a customizable simulation area, a customizable set of mobility models, and models for the propagation of the radio waves. Contrary to more popular wireless network simulators, MADHOC allows us to run the actual code of the middleware on every network node, hence taking into account not only algorithmic issues but all the implementation choices as well.

[1] http://sourceforge.net/projects/wsdl4j
[2] http://java.net/projects/saaj
[3] http://www-valoria.univ-ubs.fr/CASA/DoDWAN

7.3 Simulation environment and evaluation metrics

I conducted simulations in order to evaluate the service platform. The main objective is to assess the performance of service discovery and invocation (in terms of response time and network load) in as realistic as possible conditions.

Environment.
Network environments present different topologies and mobility behaviors that directly influence the quality of the service provision. The most widely used environment for simulations in the literature so far presents a square area with a uniform node concentration and a random way point mobility behavior. Simulations using this basic type of environment produce very good communication qualities because of its uniform mobility, and to some extent it enables predictability where any two nodes can eventually meet in some point of time. For the simulations presented in this chapter, I opted for a more realistic urban environment with buildings and mobility restrictions (see Figure 7.2). In effect, the mobility of the nodes is restricted to predefined routes between buildings, the nodes cannot access all the buildings of the simulation area, and they have a random volatility behavior.

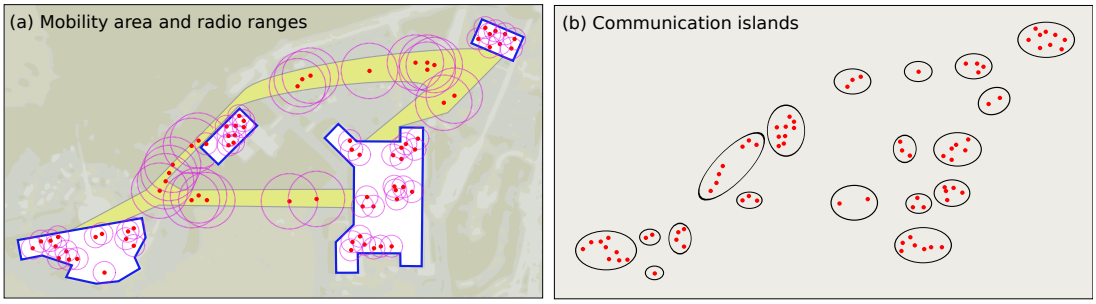


Figure 7.2: Simulation environment of a disconnected MANET. (a) The area represents a university campus where nodes communicate using Wi-Fi in ad hoc mode (radio ranges are illustrated). (b) The network presents disconnected islands of connectivity.

The simulation environment consists of a 500x900m^2 area. Nodes in the network represent users equipped with Wi-Fi capable handheld devices. Nodes live in buildings, they take roads to go from one building to another at human walking speeds (5 km/h). At any time a node may stop for a random period of time (between 1 and 6 minutes). Nodes are volatile: a node is switched on and off in a random manner. Nodes have different wireless ranges whether they are indoor (20m) or outdoor (50m). A node has affinities with some buildings and is not capable of accessing all the environment areas during a simulation run. In effect, a node stays inside its building on 70% of its movement decisions.

This environment corresponds to realistic disconnected environments with unpredictable mobility behaviors. The disconnected nature of the network is depicted in Figure 7.2(b), where nodes form separate islands of connectivity.

A service platform is deployed on each node of the network. Some nodes play the role of providers, some other nodes play the role of clients. The remaining nodes that are nei-

ther providers nor clients, can sometimes participate as simple messages relays. A node becomes a relay node if it is interested in the contents of the disseminating messages.

Evaluation metrics.
Simulations help evaluate the satisfaction of clients. Client satisfaction parameters are divided into discovery satisfaction and invocation satisfaction. Discovery satisfaction delay *DSD* is the delay between the time a client subscribes its predicate to find a suitable service and the time when this client receives a service descriptor. Invocation satisfaction delay *ISD* is the delay between the time the client publishes its request to the service and the time when it receives its response. In consequence, a client needs a delay of (*DSD* + *ISD*) in order to complete its first invocation. Every stateless invocation afterwords will only take an *ISD* delay to complete. In case of stateful invocations it takes (*DSD* + n.*ISD*) delay to complete n invocations, where ISDs are not necessarily equal.

Simulations also estimate the cost of discovery and invocation in terms of the load on the radio medium. For this purpose, I measure the number of messages sent and received by each network node, as well as the global amount of transmitted data.

In an environment of *N* nodes containing *p* provider nodes of the same business service *S1*, *r = (100 * p / N)* represents the percentage of service replication in the environment.

7.4 Discovery evaluation

The performance of discovery is directly related to the performance of the underlying communication protocol, since the providers are simply publishing their descriptors and the clients are simply subscribing in order to collect these descriptors.

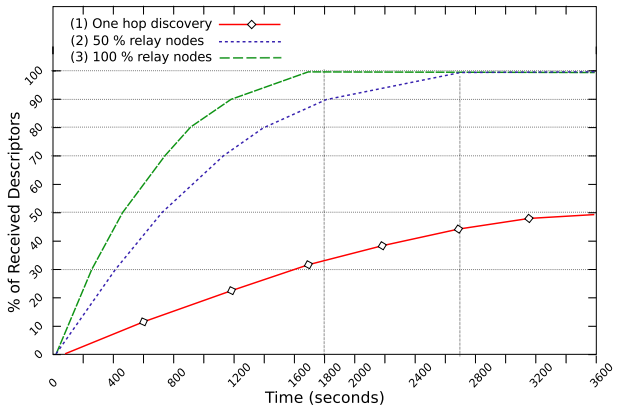
The simulation environment is populated with 80 network nodes, among which 8 play the role of service providers and 32 act as service clients. The remaining 40 nodes are neither providers nor clients but are used as simple message relays in some simulation runs. Four different business services are deployed on the providers; each service is actually deployed twice in order to obtain 8 provider nodes. Four different types of clients are present, forming four sets of eight clients targeting the same service. In this environment, the percentage of replication is r= 100 * 2 / 80 = 2.5%.

At the start of a simulation run, providers and clients are randomly distributed in the 4 buildings. And from the start, providers publish their descriptors and clients subscribe. The experiment consists in running three simulation scenarios, all with the same node mobility model but with various behaviors with respect to message dissemination:

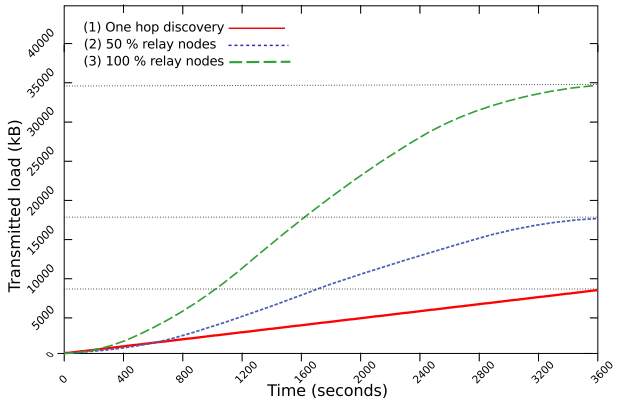
1. The first simulation scenario is used as reference. All 80 nodes have disabled their store-carry-and-forward module of the communication layer. As a consequence, no network-wide dissemination is possible. In this proximity scheme, providers and clients must be in wireless range to interact. This scenario is intended to model an approach that gives results very similar to what could be obtained with more classical one-hop broadcast-based discovery methods. Due to the absence of dissemination, providers periodically advertise their descriptors every 15 seconds.

2. In the second simulation scenario, only the 8 providers and 32 clients participate selectively in the message dissemination performed by the communication layer. The remaining 40 nodes are not interested in relaying any type of message. The low level gossiping of the communication protocol has a period of 15 seconds between catalog announcements.

3. The third simulation scenario is similar to the second, but now, the 40 nodes that are neither providers nor clients become interested in relaying messages. So we obtain 40 provider and client nodes that relay the messages and effectively consume the messages relative to service discovery and invocation they are concerned with, and 40 additional nodes just helping in the message dissemination process, regardless of the type of service involved.



(a) Client Satisfaction



(b) Network Load

Figure 7.3: Performance of discovery

The discovery satisfaction ratio is defined as the percentage of clients that have discovered the service they targeted at a given time. Figure 7.3(a) presents the results obtained for the three simulation scenarios described above. After 1800 seconds only 30% of the clients are satisfied when communications are restricted to one-hop transmissions (scenario 1). After the same period of time, the use of network-wide dissemination has a drastic impact on the speed of discovery: 90% of the clients have discovered the service they wanted when only half of the nodes (that is the providers and the clients) participate

in the dissemination (scenario 2 with 40 relays). This percentage reaches 100% when all the nodes are ready to play the role of relay (scenario 3 with 80 relays).

Resorting to network-wide dissemination for service advertisement yields a higher number of messages when compared to the one-hop broadcast-based method. However, the gossiping protocol implemented in DoDWAN has been designed so as to minimize the number and the size of the exchanged messages, thus reducing the actual load. Figure 7.3(b) plots the measured load of the network over time in the three considered scenarios. This load is expressed as the cumulative amount of data (in kB) transmitted by all nodes in the wireless medium. The figure confirms that the load in the one hop discovery increases linearly, because of the periodic advertisements of service descriptors. The curves for the scenarios involving network-wide dissemination have a higher slope but tends to flatten after a while. It can be noticed that the number of relays can be chosen to obtain a good compromise between the client satisfaction and the network load. In this example, maintaining the number of relays to 40 have induced a reasonable load while offering a fairly good satisfaction ratio.

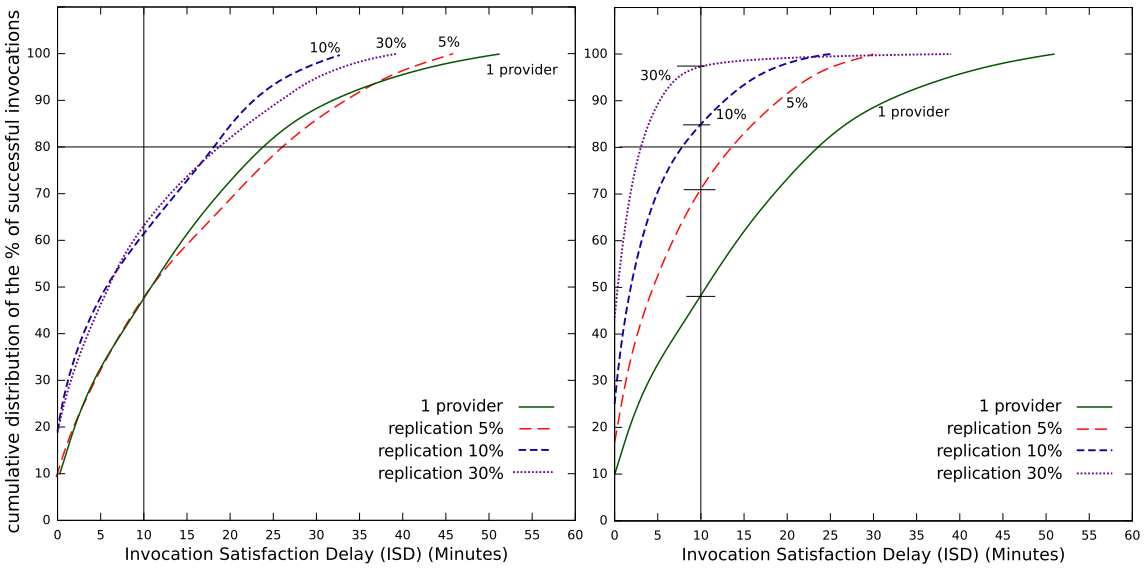
7.5 Invocation evaluation

The simulation scenario for evaluating the invocation consists of populating the environment with N=100 network nodes. The same business service is deployed with a variable replication percentage, r changes from one simulation run to another. There is only one client node seeking this business service. For every value of r, 50 runs are performed in order to obtain an averaged distribution curve for *DSD* and *ISD* client satisfaction delays. In each run, the client starts by discovering the service, and the *DSD* is measured. After discovery, the client behavior is to perform 20 invocations randomly executed throughout the duration of the simulation run, in order to plot a distribution curve for *ISD*. The network load is also measured. Measures are averaged over the 50 runs.

The following simulation results show the interest of taking advantage of service replication through the use of content-based invocations in Section 7.5.1. In addition, they show the performance of the proposed network healing solution on reducing the network load in Section 7.5.2. They also show that the service session management enhances session recovery in Section 7.5.3.

7.5.1 Effects of invoking multiple providers

The content-based approach I followed is intended to accelerate the invocation process when several providers propose the service. Figure 7.4 shows, for several service replication rates, a comparison of the invocation satisfaction delay obtained when (a) the invocation requests are sent with destination-based messages to a pre-discovered provider and (b) content-based invocation requests can reach any compatible provider and the client exploits the first received response. Each curve plots the cumulative distribution of the percentage of successful invocations against the measured average delay. As an example, in Figure 7.4-(b), we see that around 70% of the invocations are completed in less than 10 minutes when the service replication is of 5% (that is, when the service is pro-



(a) Destination-based invocations

(b) Content-based invocations

Figure 7.4: Effects of content-based invocations of multiple providers

vided by 5 of the 100 nodes). The results show that the use of content-based invocation effectively takes advantage of service replication: the more the percentage of replication the quicker the curves reaches high values, whereas replication has little effect when destination-based invocations are used. The curves in Figure 7.4-(a) show for example that, when considering a percentage of 80% of the invocations completed, the satisfaction delay passes from 24 minutes with no replication to 20 minutes with 30% replication (compared to a change from 24 to 3 minutes when using content-based invocations in the same conditions). Moreover, it is worth noticing that the gain brought about by content-based invocations is significant even for a low percentage of service replication.

In the remaining of the presented results, I consider only one value for the service replication, fixing it at a 10%. A low value has been chosen, knowing that the actual replication rate could vary a lot according to the nature of the application services. Figure 7.5 plots the cumulative distribution of satisfaction for 10% replication.

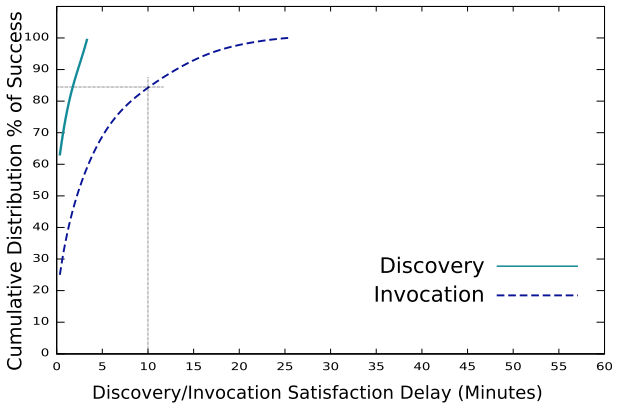


Figure 7.5: Discovery and invocation satisfactions for 10% provider replication

We can see that the client can have 100% discovery satisfaction in under 5 minutes of delay from when he wished for a service till the time he received a compatible service descriptor. Figure 7.5 also shows the cumulative distribution of succeeded invocations, 100% of succeeded invocations take less than 25 minutes to finish. In case a client has only 10 minutes to wait for his response, then he has about 85% probability of success (the invocation curve is the one found in Figure 7.4 with 10% replication).

7.5.2 Network healing

Provider replication increases the network load by adding extra response messages. The healing protocol is used to clean up the network from redundant and unneeded messages. Simulations show the effectiveness of this healing by measuring the network load with and without healing. Simulations also show that the healing does not influence the satisfaction of clients.

In order to show the effectiveness of the full healing solution proposed in the previous chapter. I used content-based invocations in four simulation scenarios, and measured their impact on the number of messages sent in the network as well as on the amount of transferred data.

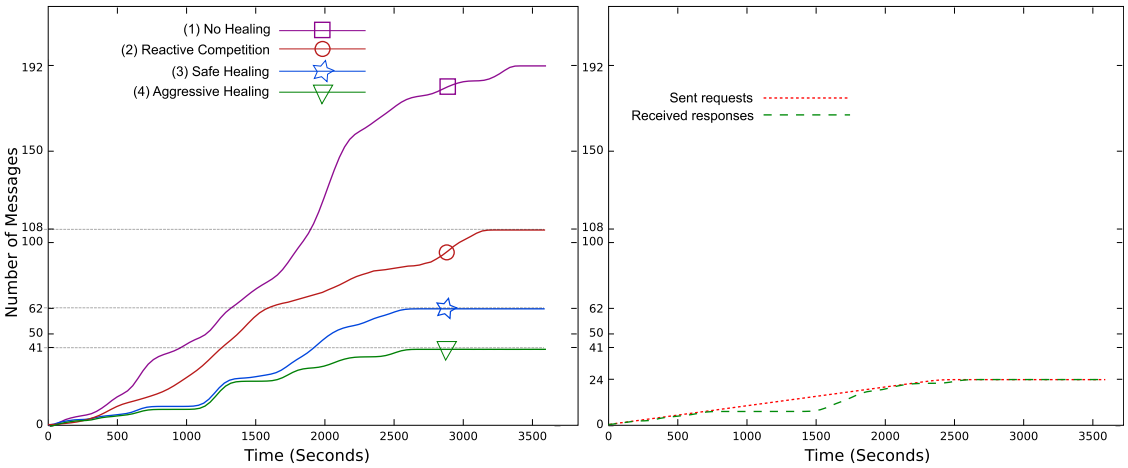
1. The first scenario does not use any healing, and is used as reference.
2. The second scenario puts in place the reactive competition (RC) mechanism performed at provider nodes.
3. The third scenario uses the safe healing type. It includes the reactive competition mechanism and adds healing messages initiated by the client. After receiving a response, the client initiates the healing of its request and corresponding response messages from the network.
4. The fourth scenario uses the aggressive healing type. It includes the reactive competition mechanism and adds healing messages initiated by the providers themselves.

Figure 7.6(a) shows the cumulative distribution of the number of responses sent by the providers against time when applying the four scenarios.

Table 7.1 shows the response reduction ratio of all scenarios. In comparison to the first scenario, the number of sent responses is drastically reduced in scenarios 2, 3, and 4 (by a factor of up to 4.7).

r=10%	(1) No Healing	(2) Reactive Competition	(3) Safe Healing	(4) Aggressive Healing
Reduction Ratio	192/192= 1	192/108= 1.7	192/62= 3	192/41= 4.7

Table 7.1: Response reduction



(a) Message healing

(b) Corresponding requests/responses

Figure 7.6: Performance of healing

Interestingly, although the aggressive healing may theoretically delay useful responses and even prevent them from arriving at the client, the results for the considered scenario do not reveal such a loss. This is illustrated by the two dotted lines in Figure 7.6(b), which represent the cumulative number of requests sent by the client and the cumulative number of responses effectively received by this client (with aggressive healing applied): responses are, as expected, slightly delayed, but the client eventually receives them all. Furthermore, Figure 7.7 compares the client satisfaction ratio between scenarios 1 and 4, responses are delayed in scenario 4 but they are not lost.

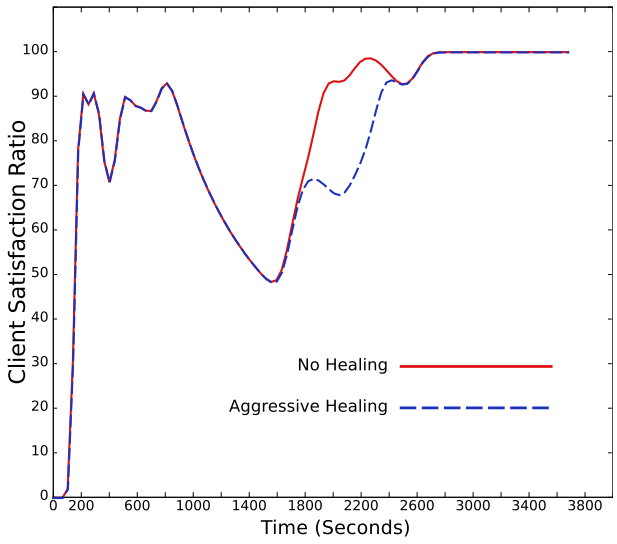
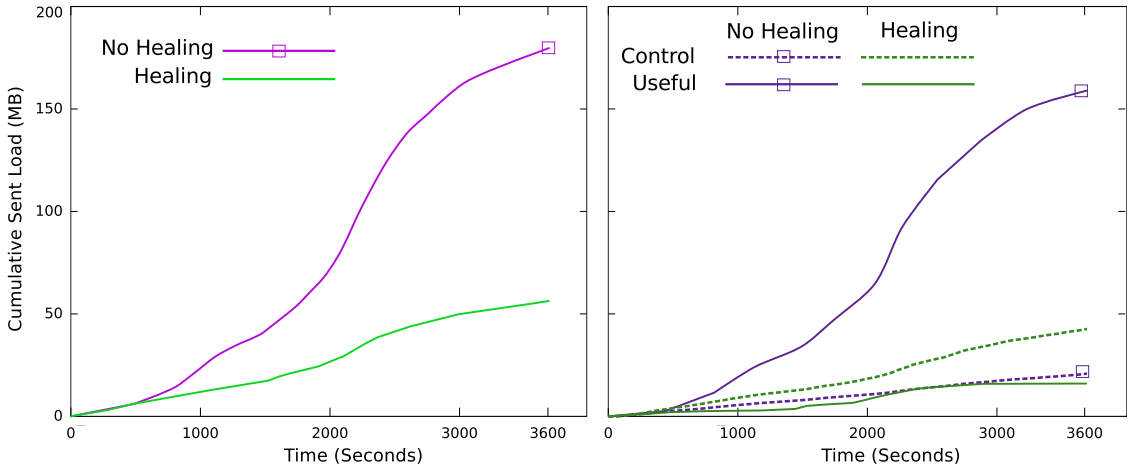


Figure 7.7: Ratio of received responses over requests

When considering the network load, the impact of network healing is even more important. Extra healing messages are added but they have a small footprint (≃200 Bytes) and they allow saving requests and responses (which may be of relatively large size). In practice, the volume of the request and responses saved can largely compensate the vol-



(a) Global Network Load

(b) Detailed Load: Useful and Control

Figure 7.8: Cumulative load sent into the network medium

ume of extra control messages. This is confirmed by the results given in Figure 7.8. This figure plots the cumulative network load sent by all the nodes into the network medium when no healing is applied as well as when the full healing is applied (scenarios 1 and 4). The network load is divided in two values: the cumulative useful payload (i.e. the requests and responses), and the cumulative size of control messages (at the service and the communication level, including extra control messages due to healing). We considered an average size of 26 kB for requests and responses. As expected, the amount of control data slightly increases when the healing is activated, but this extra cost is very small compared to the amount of data that is required for redundant requests and responses when no healing is applied: for the entire experiment, 27 MB of control messages are added but 142 MB of useful payload are saved.

7.5.3 Session recovery for stateful invocations

I compared the efficiency of the session recovery mechanism based on content-based invocation to a more classic approach used in fully connected MANETs. In this latter approach, communication can only take place inside an island, using multi-hop transmissions that exploit a routing algorithm. I measured the time taken for session recovery in a simulated scenario where a client tries to maintain a session with a service provider, within which it periodically sends invocation requests with a 5-minute timeout. The session is considered broken if the response has not arrived 5 minutes after the request is sent. Each of the providers are randomly turned off and back on 5 times during a simulation run.

In the routed scheme, the client iterates on the following steps: (1) it discovers a list of providers that propose the service it wants by broadcasting a discovery request in its entire communication island and by waiting for a discovery response; (2) it opens a session with the first provider that responds; (3) it performs a sequence of invocations to the chosen provider until the session is broken. The scenario with the proposed session recovery scheme is simpler as the discovery phase has not to be repeated: after having

discovered the existence of the wanted service, the client iterates on the following steps: (1) it performs a first content-based invocation, opening a session with the first provider that responds; (2) it performs a sequence of destination-based invocations to this provider until the session is broken. Figure 7.9 shows the cumulative distribution of the percentage of session recovery in the two schemes.

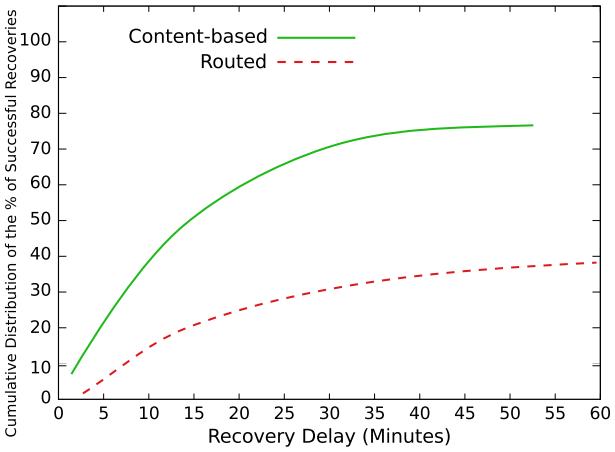


Figure 7.9: Session management.

The content-based session recovery scheme clearly outperforms the routed scheme. For example, with the content-based approach, 60% of the sessions are recovered in less than 20 minutes whereas only 25% of the sessions are recovered in less that this same duration with the routed scheme. Two main reasons explain this: the discovery phase is performed only once, and the communication method is not limited to the communication island so it allows the client to reach any provider in the network when trying to re-establish a session.

7.6 Conclusion

This chapter presented the implementation of the service platform as a middleware architecture. The middleware consists of a service layer (DiSWAN) and a communication layer (DoDWAN). Even though the platform separates the two layers to provide a clear distinction between service layer's protocols and communication layer's protocols, the agents of the service layer have direct access on message manipulation, which enables a better cross-layer coordination. Using this middleware implementation, I conducted simulations in a disconnected MANET environment in order to evaluate the discovery and invocation protocols. The main objective is to assess the performance in terms of response time and network load. The performance of discovery is directly related to those of the underlying communication layer. Simulations measure the client satisfaction with discovery, and the time delays needed for this discovery. Simulations also measure the client satisfaction with invocation. They show the benefits of content-based invocations in providing faster response times and better satisfaction (when invoking a business service independently of its providers). Simulations also show the drastic reduction of redundant and leftover messages when applying the healing mechanisms, without influ-

encing the proper execution of the invocation. In addition, simulations showed that when sessions are needed, the proposed solution provides better session recovery in comparison to traditional routing techniques.

Concerning the simulation itself, even though I tried to create a university campus environment with as realistic as possible conditions, still node behavior could be very different in the real world. Moreover, I actually conducted more simulations by varying many parameters. For example, when letting all the nodes move freely in all network areas, the simulation produces better results from the ones presented in this chapter. And when nodes are restricted to their buildings with very limited movement between buildings, the simulation produces worse results with a high rate of delivery failures. The results presented in this chapter restrict nodes to their buildings on 70% of their movement decisions, which lets a number of the nodes at a given time move between buildings in order to enable network-wide dissemination. In addition, the presented simulations may be specific to the university campus environment, and the performance of the platform may change in other types of environments.

8 Conclusions and Perspectives

8.1 Conclusions

This thesis investigated the benefits and challenges of using the service-oriented computing in disconnected mobile ad hoc network (MANET) environments. My overall objective was to build a service middleware platform for mobile nodes that supports the execution of service-oriented applications in disconnected MANETs.

In order to design this service platform, I studied the service-oriented computing approach and how communications can be provided in the targeted environments. I presented the state of the art of both communications and service-oriented systems in mobile ad hoc networks.

MANETs are spontaneously formed networks that do not need any form of infrastructure. In disconnected MANETs, the network contains disconnected islands between which traditional routing protocols do not work. In such disconnected networks, communications must be tolerant to disruptions, and protocols must use a store-carry-and-forward mechanism to enable network-wide communications.

The service-oriented computing (SOC) model seems suited for ad hoc environments because it emphasizes the decoupled nature of its entities. Effectively, the decoupling between the client and the provider entities becomes essential in mobile environments with a fluctuating availability of providers, and where end-to-end communications are not guaranteed. Still, in existing service-oriented systems, providers are usually supposed to be always available, as it is the case for example in Web services. Providers are also assumed to be continuously reachable in wireless mobile environments, where local connected networks can be created using Wi-Fi hotspots, or by creating ad hoc networks using proximity one-hop or routed multi-hop protocols. Although the service-oriented approach seems relevant for disconnected MANETS, implementing distributed services for such networks still poses several challenges. Not only network-wide communication features must be provided, in spite of constant network fragmentation, but aspects such as the unpredictable reachability of the providers, or potential communication delays, must be taken into account at the service level.

I proposed a service platform implemented as a middleware of two layers: a communication protocol resides in the lower layer, and discovery and invocation protocols reside in the higher layer.

Due to the high communication constraints imposed by disconnected environments,

I used an opportunistic and content-driven protocol (DoDWAN) to implement the lower layer of the platform. Chapter 4 described the inner workings of the protocol, implementing a message store-carry-and-forward paradigm, opportunistic gossiping, and content-based matching. A publish/subscribe API interfaces the protocol with the upper service layer. The communication challenges of disconnected MANETs impose that the communication layer enable the decoupling at the three time, synchronization, and space dimensions. With unpredictable availability, there is a high probability that two service entities wanting to communicate are not active at the same time or that they reside in different parts of the network (islands). In addition, a network node cannot be blocked after producing a message, and likewise it cannot be blocked while waiting to receive a message. Furthermore, two interacting nodes do not need to know and hold references to each other. The producer of information does not know how many nodes are consuming this information. Likewise, the consumer of information does not know how many producers it is getting its information from. These communication decoupling challenges are well met by the DoDWAN communication protocol.

I proposed a discovery protocol for the service layer of the platform. Chapter 5 presented the life-cycles of the client and provider nodes. It also presented the elements that construct the discovery protocol, which are the description and advertisement at the provider side, and collection and selection at the client side. Service description uses functional and non-functional properties. Providers advertise their services by adding context information to the descriptions. Discovery is based on a peer-to-peer only architecture using local directories at each node, where a client collects only interesting services. The discovery relies on the content-based message dissemination of the underlying communication layer, giving an aspect of cross-layer design. After collecting interesting service, the client selects the most appropriate one to invoke.

The main purpose is to improve the loose coupling between the providers and the consumers of services. The more knowledge about the provider that a client has to remember and keep up-to-date, the more ties exist between the provider and client entities, and it becomes harder to handle a change of availability of the provider. In case the provider is no longer available, the client in the worst case needs to rediscover other providers, hence reducing the performance of the service provision. Content-based invocation solutions help increase loose coupling and enhance the system's performance.

I also proposed invocation solutions for the service layer. Chapter 6 assumes that in mobile environments, there might be a multitude of providers offering the same business service. Content-based invocations are used to benefit from this service replication. In the default remote invocation behavior, a client creates a functional request according to a discovered service provider. The client publishes its request message to the attention of the discovered provider, in addition to other compatible providers. Compatible providers can receive the client's request. A provider creates a response message and publishes it back to the client. According to the client's needs, it may get satisfied when receiving the first response or it may collect multiple responses to be satisfied. The invocation completes when the client gets satisfied. After being satisfied, the client can initiate the healing of residual request and response messages from the network. The healing of response messages can also be initiated by the providers themselves according to the priority of the response messages. The client also has the possibility to restrict this default behavior to specific provider destinations. In worst-case scenarios, where a client

cannot collect any interesting service, blind invocation requests are used to overcome the discovery failure. In addition, the chapter presented stateful remote invocation solutions using session management and recovery.

And finally Chapter 7 presented the implementation of the service platform as a middleware architecture. The middleware consists of a service layer (DiSWAN) and a communication layer (DoDWAN). The platform separates the two layers to provide a clear distinction between service layer's protocols and communication layer's protocols. Still, the agents of the service layer have direct access to message creation and manipulation. Using this middleware implementation, I conducted simulations in a disconnected MANET environment, using as realistic as possible conditions in order to evaluate the discovery and invocation protocols. The main objective was to assess the performance of the middleware platform in terms of response time and network load. The performances of discovery is directly related to those of the underlying communication layer. Simulations measured the client satisfaction with discovery, and the time delays needed for this discovery. Simulations also measured the client satisfaction with invocation. They confirmed the benefits of content-based invocations in providing faster response times and better satisfaction (when invoking a service independently of its providers). Simulations also showed the drastic reduction of redundant and leftover messages when applying the healing mechanisms, without influencing the proper execution of the invocation. In addition, simulations showed that when sessions are needed, the proposed solution provides better session recovery in comparison to traditional routing techniques.

8.2 Perspectives

In the following, I enumerate some interesting perspectives for future work.

Security

This work did not try to address security issues in the design of the service platform. In MANET environments, the most obvious measure is to ensure the authenticity of messages disseminating in the network. Authenticity is hard to implement in mobile environments, unless closed communities are created in order for them to identify each of their members, which tends to go against the nature of mobile networks. Still, some simple measures could be implemented. For example a sender could include its signature in its messages. Furthermore, the payload of messages could be encrypted for interactions between trusted friends, which does not hinder the dissemination of messages based on the contents of their headers, yet relay nodes might not accept to relay encrypted messages. Another example might be using authorized service access at the provider nodes to control client invocations, which can hinder the performance of content-based invocations.

Semantic descriptions and ontologies

This thesis studies the challenges of disconnected mobile ad hoc networks from the service provision point of view. It uses solutions for network-wide communication, and proposes solutions for service discovery and invocation. One of the key novelties is enabling a service provider to accept the reception of a request initially created according to

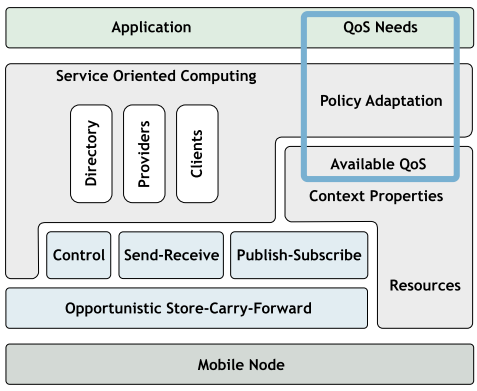
the description of another provider. I believe it represents a transition in the service provision behavior from the traditional discovery-invocation technique to a an invocation only technique.

Full-blown ontology languages provide specific and large domain ontologies and spoken language ontologies. Yet, mobile devices are usually not capable of providing the large storage space nor the computational power needed for such semantic systems. Ontologies might eventually allow semantic-only invocations, where no a priori discovery is needed. Even though the blind request mechanism proposed in this document can somehow bypass discovery, it is based on keyword matching and does not resort to any ontologies.

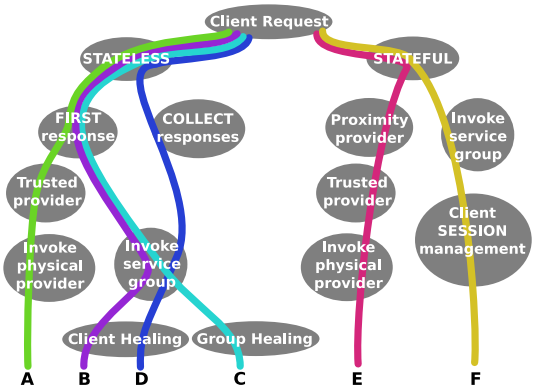
Ontology systems need the ontology language to be known by all the entities of the network. A possible solution might be to use reduced size specific domain ontologies that would be distributed to all nodes. Yet, synchronization problems might occur if this ontology was updated on the service side and not on the client side. One can imagine the extension for pure ad hoc networks, where nodes can connect to the internet from time to time in order to install or synchronize their ontologies.

Adaptive service provision

The proposed service platform provided adaptation mechanisms namely at the selection stage where a client agent selects the best provider according to the context properties. This adaptation can be further enhanced in order to enable a better service provision, where the needed quality of service properties are adapted to the available quality of service properties. Available QoS can represent hardware resources (e.g. processing, storage), and network resources (how well the node can communicate to other nodes), and context properties (e.g. location information, availability). Applications can declare their QoS needs (e.g. needed response time, use trusted providers). Automated adaptation between these needed and available QoS gives place to a better provision process. Figure 8.1(a) shows how quality of service adaptation between available and needed QoS might be implemented in the service middleware. Figure 8.1(b) shows some use cases of adaptation.



(a) QoS adaptation can be added to the middleware



(b) Adaptation use case examples

Figure 8.1: Quality of service adaptation perspectives

Another form of adaptation is service-level adaptation: depending on the requester's requirements, the service provider's capabilities, and the dynamic runtime conditions such as service load. For example, a provider receives incoming requests and enables prioritization according to predefined classifications, the request is then placed on the appropriate priority queue. Another example can be that a provider makes sure the number of requests per customer is within a predefined limit, and exceeding requests are assigned a low priority. This automated resource management and adaptation can be very challenging in disconnected MANETs.

Real world experiments

Recent experimentation campaigns were conducted at the campus of the "Université de Bretagne Sud" (DoDWAN-Expe[1]). These campaigns collected the traces of contacts between network nodes as well as the network traffic produced by the DoDWAN protocol. The evaluation of the service platform can be enhanced when using these real contact traces in the simulation environment. Moreover, future works should perfect the implementation and the API provided by DiSWAN to higher layers in order to create applications and conduct experiments in real world environments.

[1] http://www-valoria.univ-ubs.fr/CASA/DODWAN-EXPE/

Personal Publications

1. Yves Mahéo and Romeo Said. Service Invocation over Content-Based Communication in Disconnected Mobile Ad Hoc Networks. In *24th International Conference on Advanced Information Networking and Applications (AINA'10)*, pp. 503-510, Perth, Australia, April 2010.

2. Romeo Said and Yves Mahéo. Toward a Platform for Service Discovery and Invocation in Disconnected Mobile Ad Hoc Networks. In *International Conference On Embedded and Ubiquitous Computing (EUC 2008)*, Shanghai, China, December 2008.

3. Nicolas Le Sommer, Romeo Said, and Yves Mahéo. A Proxy-based Model for Service Provision in Opportunistic Networks. In *6th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC'08)*, Louvain, Belgium, December 2008.

4. Yves Mahéo, Romeo Said, and Frédéric Guidec. Middleware Support for Delay-Tolerant Service Provision in Disconnected Mobile Ad Hoc Networks. In *Workshop on Java and Components for Parallelism, Distribution and Concurrency at IPDPS'08*, Miami, FL, USA, April 2008.

References

[1] Changling Liu and Jörg Kaiser. A Survey of Mobile Ad Hoc network Routing Protocols. Technical report, University of Magdeburg, 2005.

[2] Kevin Fall and Stephen Farrell. DTNRG, Delay Tolerant Networking Research Group, IRTF Internet Research Task Force. www.dtnrg.org.

[3] GSM Alliance. GSM Technology. www.gsmworld.com.

[4] IEEE Standard for Information Technology. Telecommunications and information exchange between systems- local and metropolitan area networks- specific requirements- part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications. *ANSI/IEEE Std 802.11, 1999 Edition (R2003)*, pages i –513, 2003.

[5] IEEE Standard for Information Technology. Telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements. - part 15: Wireless medium access control (mac) and physical layer (phy) specifications for wireless personal area networks (wpans). *IEEE Std 802.15-2002)*, 2002.

[6] Stephan OLARIU and Michele C. WEIGLE, editors. *Vehicular Networks: From Theory to Practice*. Chapman & Hall/CRC Computer and Information Science Series, 2009.

[7] Bartosz Wietrzyk, Milena Radenkovic, and Ivaylo Kostadinov. Practical MANETs for Pervasive Cattle Monitoring. In *ICN*, pages 14–23, 2008.

[8] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 96–107, New York, NY, USA, 2002. ACM.

[9] Alex Pentland, Richard Fletcher, and Amir Hasson. Daknet: Rethinking connectivity in developing nations. *IEEE Computer*, 37(1):78–83, 2004.

[10] Thomas Clausen, Christopher Dearlove, and Philippe Jacquet. The optimized link state routing protocol version 2, IETF, draft-ietf-manet-olsrv2-11, October 2010.

[11] Ian Chakeres and Charles Perkins. Dynamic manet on-demand (dymo) routing, IETF, draft-ietf-manet-dymo-21, January 2011.

[12] Zhensheng Zhang and Qian Zhang. Delay/disruption tolerant mobile ad hoc networks: latest developments: Research articles. *Wireless Communications and Mobile Computing*, 7(10):1219–1232, 2007.

[13] Zhensheng Zhang. Routing in Intermittently Connected Mobile Ad Hoc Networks and Delay Tolerant Networks: Overview and Challenges. *IEEE Communications Surveys and Tutorials*, 8(1):24–37, January 2006.

[14] Luciana Pelusi, Andrea Passarella, and Marco Conti. Opportunistic Networking: Data Forwarding in Disconnected Mobile Ad Hoc Networks. *IEEE Communications Magazine*, November 2006.

[15] Kevin Fall, Wei Hong, and Samuel Madden. Custody transfer for reliable delivery in delay tolerant networks. Technical report, Intel Research Berkeley, 2003.

[16] Amin Vahdat and David Becker. Epidemic routing for partially connected ad hoc networks. Technical report, Duke University, 2000.

[17] Mirco Musolesi, Cecilia Mascolo, and Stephen Hailes. Emma: Epidemic messaging middleware for ad hoc networks. *Personal Ubiquitous Computing*, 10(1):28–36, 2005.

[18] Khaled A. Harras, Kevin C. Almeroth, and Elizabeth M. Belding-Royer. Delay tolerant mobile networks (dtmns): Controlled flooding in sparse mobile networks. In *In IFIP Networking Conference*, Ontario, CANADA, May 2005.

[19] Thrasyvoulos Spyropoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. Spray and wait: an efficient routing scheme for intermittently connected mobile networks. In *WDTN '05: Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking*, pages 252–259, New York, NY, USA, 2005. ACM.

[20] Chiara Boldrini, Marco Conti, and Andrea Passarella. Context and resource awareness in opportunistic network data dissemination. In *The Second IEEE WoWMoM Workshop on Autonomic and Opportunistic Communications (AOC 2008)*, Newport Beach, CA, USA, June 2008.

[21] Anders Lindgren, Avri Doria, and Olov Schelen. Probabilistic routing in intermittently connected networks. In Petre Dini, Pascal Lorenz, and Jose Neuman de Souza, editors, *Service Assurance with Partial and Intermittent Resources*, volume 3126 of *Lecture Notes in Computer Science*, pages 239–254. Springer Berlin Heidelberg, 2004.

[22] Antonio Carzaniga and Alexander L. Wolf. Content-based Networking: A New Communication Infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in LNCS, pages 59–68, Scottsdale, Arizona, October 2001. Springer-Verlag.

[23] Paola Costa, Mirco Musolesi, Cecilia Mascolo, and Gian Petro Picco. Adaptive Content-based Routing for Delay-tolerant Mobile Ad Hoc Networks. Technical report, UCL, August 2006.

[24] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2), 2003.

[25] Julien Haillot and Frédéric Guidec. A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks. In *IEEE 22nd Int. Conf. on Advanced Information Networking and Applications (AINA'08)*, Okinawa, Japan, March 2008.

[26] Julien Haillot and Frédéric Guidec. A Protocol for Content-Based Communication in Disconnected Mobile Ad Hoc Networks. *Journal of Mobile Information Systems*, 6(2):123–154, 2010.

[27] Thomas Erl. *SOA Principles of Service Design*. Prentice Hall, 2007. ISBN-10: 0132344823.

[28] Gregory Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[29] Sun Microsystems, Inc. RPC: Remote Procedure Call, Protocol Specification Version 2, RFC 1057, June 1988.

[30] ORACLE. JAVA Remote Method Invocation. http://www.oracle.com.

[31] Microsoft Corporation. DCOM. http://www.microsoft.com.

[32] Apple. Objective-C. http://developer.apple.com.

[33] Pyro. Python remote objects. http://www.xs4all.nl/ irmen/pyro3/.

[34] Object Management Group OMG. CORBA. http://www.omg.org.

[35] ORACLE. JAVA Enterprise Edition. http://www.oracle.com.

[36] Microsoft Corporation. .NET Remoting. http://msdn.microsoft.com.

[37] Philippe Merle and Jean-Bernard Stefani. A formal specification of the Fractal component model in Alloy. Research Report RR-6721, INRIA, 2008.

[38] W3C CONSORTIUM. SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, April 2007. http://www.w3.org.

[39] W3C CONSORTIUM. Web services architecture. W3C Working Group Note, February 2004. http://www.w3.org.

[40] ORACLE. JSR 166: Concurrency Utilities. http://jcp.org/en/jsr/detail?id=166.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. AddisonWesley, 1994. ISBN 0-201-63361-2.

[42] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7:80–112, 1985.

[43] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 15, no. 3:279–328, July 2006.

[44] Sun Microsystems. Javaspaces specification, 1999.

[45] J. Paul Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994. ISBN 0-442-01771-5.

[46] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A case for message oriented middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 1–18, London, UK, 1999. Springer-Verlag. 3-540-66531-5.

[47] Apache Software Foundation. Qpid open source advanced message queuing protocol. http://qpid.apache.org.

[48] ORACLE. Java message service JMS specification.

[49] Keneth Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM.

[50] TIBCO. TIBCO Rendezvous. http://www.tibco.com.

[51] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 219–227, 2000.

[52] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[53] Object Management Group OMG. DDS Data Distribution Service for Real-time Systems. http://www.omg.org.

[54] OSGi Alliance. OSGi service platform, Core Specification, Release 4, Version 4.2, September 2009. http://www.osgi.org.

[55] Jan S. Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: distributed applications through software modularization. In *Middleware 07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 1–20, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[56] ORACLE. GlassFish Server Open Source Edition. http://glassfish.dev.java.net.

[57] JBoss Enterprise, Red Hat. JBoss Application Server. http://www.jboss.org.

[58] ObjectWeb Consortium. JOnAS Application Server. http://wiki.jonas.ow2.org.

[59] Dave Chappell. *Enterprise Service Bus*. OReilly, June 2004. ISBN 0-596-00675-6.

[60] ORACLE. The Open Enterprise Service Bus. http://open-esb.dev.java.net.

[61] IBM. WebSphere Enterprise Service Bus. www.ibm.com.

[62] OASIS. Web Services Business Process Execution Language (WSBPEL), 2006. www.oasis-open.org.

[63] ORACLE. Oracle Fusion Middleware. www.oracle.com.

[64] SAP. Service-Oriented Architecture. www.sdn.sap.com.

[65] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[66] OASIS. Service Component Architecture SCA. SCA specifications, March 2007. http://www.oasis-opencsa.org/sca.

[67] W3C CONSORTIUM. Web Services Description Language (WSDL) Version 2.0. W3C Recommendation, June 2007. www.w3.org.

[68] W3C CONSORTIUM. Semantic Annotations for WSDL and XML Schema. W3C Recommendation, August 2007. http://www.w3.org.

[69] W3C CONSORTIUM. SOAP over Java Message Service 1.0. W3C Candidate Recommendation, June 2009. http://www.w3.org.

[70] OASIS. UDDI Version 3.0.2. UDDI Spec Technical Committee Draft, November 2004. http://uddi.org.

[71] W3C CONSORTIUM. Web Services Choreography Description Language Version 1.0. W3C Candidate Recommendation, November 2005. http://www.w3.org.

[72] Apache Software Foundation. Jini technology, Apache River. http://www.jini.org.

[73] Erik Guttman, Charles Perkins, John Veizades, and Michael Day. Service Location Protocol, Version 2. IETF RFC 2608, June 1999.

[74] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. Internet Draft, February 2011.

[75] P. Mockapetris. Domain Name Service. Standard, November 1987. RFC 1034.

[76] UPnP Forum. UPnP Device Architecture, Version 1.1, October 2008. http://www.upnp.org.

[77] Varon Y. Goland and Ting Cai and Paul Leach and Ye Gu. Simple service discovery protocol 1.0. Internet draft, April 1999.

[78] Josh Cohen and Sonu Aggarwal. General Event Notification Architecture. Internet draft, July 1998.

[79] Salutation Consortium. Salutation, October 2003. http://salutation.org.

[80] Bluetooth Special Interest Group. Specification of the Bluetooth System, Version 2.1 + EDR, Service Discovery Protocol, July 2007. http://www.bluetooth.com.

[81] IGRS Alliance. IGRS Standard v1.0. http://www.igrs.org/indexen.aspx.

[82] ECHONET CONSORTIUM. ECHONET Specification. http://www.echonet.gr.jp/english.

[83] Reto Hermann, Dirk Husemann, Michael Moser, Michael Nidd, Christian Rohner, and Andreas Schade. DEAPspace - Transient ad hoc networking of pervasive devices. *Computer Networks*, 35(4):411–428, March 2001.

[84] Sumi Helal, Nitin Desai, Varun Verma, and Choonhwa Lee. Konark : Service Discovery and Delivery Protocol for Ad-hoc Networks. In *3rd IEEE Conf. on Wireless Communication Networks (WCNC)*, New Orleans, USA, March 2003.

[85] Dipanjan Chakraborty, Anupam Joshi, Yelena Yesha, and Tim Finin. Toward Distributed service discovery in pervasive computing environments. *IEEE Transactions on Mobile Computing*, 5(2):97– 112, February 2006.

[86] W3C CONSORTIUM. OWL Web Ontology Language. W3C Recommendation, February 2004. http://www.w3.org.

[87] Dipanjan Chakraborty, Anupam Joshi, and Yelena Yesha. Integrating service discovery with routing and session management for ad-hoc networks. *Ad Hoc Networks*, 4(2):204–224, March 2006.

[88] Françoise Sailhan and Valérie Issarny. Scalable Service Discovery for MANET. In *Int. Conf. on Pervasive Computing and Communications (PerCom'2005)*, Hawai, USA, March 2005. IEEE Press.

[89] Celeste Campo, Mario Munoz, Jose Carlos Perea, Andreas Marin, and Carlos Garca-Rubio. PDP and GSDL: A New Service Discovery Middleware to Support Spontaneous Interactions in Pervasive Systems. In *IEEE Middleware Support for Pervasive Computing (PerWare 2005)*, Hawaii, March 2005.

[90] Li Li and Louise Lamont. A Lightweight Service Discovery Mechanism for Mobile Ad Hoc Pervasive Environment Using Cross-Layer Design. In *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*, pages 55–59, Washington, DC, USA, 2005. IEEE Computer Society.

[91] Radu Handorean, Rohan Sen, Greg Hackmann, and Gruia-Catalin Roman. Supporting Predictable Service Provision in MANETs via Context Aware Session Management. *International Journal of Web Services Research*, (3):1–26, 2006.

[92] Christopher N. Ververidis and George C. Polyzos. Service discovery for mobile ad hoc networks: A survey of issues and techniques. *IEEE Communications Surveys and Tutorials*, 10(3):30–45, 2008.

[93] Adnan Noor Mian, Roberto Baldoni, and Roberto Beraldi. A Survey of Service Discovery Protocols in Multihop Mobile Ad Hoc Networks. *IEEE Pervasive Computing*, 8:66–74, 2009.

[94] Seyed Amin Hosseini Seno, Rahmat Budiarto, and Tat-Chee Wan. Survey and new Approach in Service Discovery and Advertisement for Mobile Ad hoc Networks. *International Journal of Computer Science and Network Security*, 7(2):275–284, 2007.

[95] Chunglae Cho and Duccki Lee. Survey of Service Discovery Architectures for Mobile Ad hoc Networks. Term paper, Mobile Computing, CEN 5531, Dept. Computer and Information Science and Eng, Univ. Florida, Fall, 2005.

[96] Rajeev Koodli and Charles Perkins. Service discovery in on-demand ad hoc networks. IETF Internet draft, October 2002.

[97] Gertjan P. Halkes, Aline Baggio, and Koen Langendoen. A simulation study of integrated service discovery. In *First European Conference on Smart Sensing and Context, EuroSSC*, pages 39–53, 2006.

[98] Jose Luis Jodra, Maribel Vara, Jose Ma Cabero, and Josu Bagazgoitia. Service discovery mechanism over olsr for mobile ad-hoc networks. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, pages 534–542, Washington, DC, USA, 2006. IEEE Computer Society.

[99] Paal E. Engelstad, Yan Zheng, Rajeev Koodli, and Charles E. Perkins. Service Discovery Architectures for On-Demand Ad Hoc Networks. *Ad Hoc and Sensor Wireless Networks*, 1, 2006.

[100] René Meier, Vinny Cahill, Andronikos Nedos, and Siobhán Clarke. Proximity-Based Service Discovery in Mobile Ad Hoc Networks. In *5th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'05)*, volume 3543 of *LNCS*, Athens, Greece, June 2005. Springer.

[101] Alex Varshavsky, Bradley Reid, and Eyal de Lara. A cross-layer approach to service discovery and selection in MANETs. In *2nd IEEE Int. Conf. on Mobile Ad-hoc and Sensor Systems (MASS 05)*, Washington, USA, November 2005. IEEE Press.

[102] Vincent Lenders, Martin May, and Bernhard Plattner. Service Discovery in Mobile Ad Hoc Networks: A Field Theoretic Approach. In *International Symposium on a World of Wireless, Mobile, and Multimedia Networks (WoWMoM 2005)*, Taormina, Italy, June 2005.

[103] Michael Klein, Birgitta Konig-Ries, and Obreiter Philipp. Service Rings - A Semantic Overlay for Service Discovery in Ad hoc Networks. pages 180–185, 2003.

[104] Michael Klein, Birgitta Konig-Ries, and Obreiter Philipp. Lanes: A Lightweight Overlay for Service Discovery in Mobile Ad Hoc Networks. In *3rd Workshop Applications and Services in Wireless Networks (ASWN 03)*, July 2003.

[105] Jerry Tyan and H. Mahmoud, Qusay. A network layer based architecture for service discovery in mobile ad hoc networks. In *17th Ann. IEEE Canadian Conf. Electrical and Computer Eng. (CCECE 04)*, volume 3, pages 1379–1384, Niagara Falls, Canada, May 2004. IEEE Press.

[106] Andronikos Nedos, Kulpreet Singh, and Siobhán Clarke. Service*: Distributed Service Advertisement for Multi-Service, Multi-Hop MANET Environments. In *7th IFIP Int. Conf. on Mobile and Wireless Communication Networks (MWCN'05)*, Marrakech, Morocco, September 2005.

[107] Lionel Touseau, Didier Donsez, and Walter Rudametkin. Towards a sla-based approach to handle service disruptions. In *Proc. of 5th IEEE International Conference on Services Computing (SCC 2008), Research track, July 8-11, 2008, Honolulu, Hawaii*, 2008.

[108] Karl Weick. Educational organizations as loosely coupled systems. *Administrative Science Quarterly*, 21:1–19, 1976.

[109] Thomas Erl. *SOA Design Patterns*. Prentice Hall PTR, 2009. ISBN-10: 0136135161.

[110] Luc Hogie, Pascal Bouvry, and Frédéric Guinand. The MADHOC simulator. http://www-lih.univ-lehavre.fr/~hogie/madhoc.

Résumé

Les réseaux mobiles ad hoc (MANETs) se forment spontanément à partir de terminaux mobiles qui communiquent en utilisant des interfaces sans fil à faible portée (e.g. Wi-Fi, Bluetooth). Dans la plupart des réseaux ad hoc déployés dans des conditions réelles, ces terminaux mobiles peuvent être volatiles et distribués de manière clairsemée, et former par conséquent des îlots de connectivité qui évoluent continuellement. Dans cette thèse, je considère spécifiquement ce type de réseaux, qui sont appelés MANETs discontinus. Les communications dans les MANETs discontinus présentent un défi, parce que les protocoles de routage conçus pour les MANETs connexes ne fonctionnent pas dans de tels réseaux. L'approche *store-carry-and-forward* est proposée depuis quelques années pour palier les discontinuités du réseau. Elle permet à un nœud de stocker temporairement un message, afin de le transmettre plus tard quand les conditions deviennent favorables. La mobilité des nœuds devient alors un avantage, en facilitant la propagation des messages d'un îlot vers un autre.

L'approche orientée services semble appropriée pour mettre en œuvre des applications dans les MANETs discontinus à cause de la nature découplée des entités qu'elle met en jeu. En effet, le découplage entre un client et un fournisseur de service est essentiel dans un environnement où la disponibilité des fournisseurs est fluctuante, et où les communications de bout en bout ne sont pas garanties. Malgré ce découplage, dans les systèmes orientés services existants les fournisseurs sont souvent supposés stables et toujours accessibles. Bien que l'approche orientée services semble appropriée dans les MANETs discontinus, la mise en œuvre de services distribués nécessite des communications dans l'ensemble du réseau, malgré la fragmentation de celui-ci. En outre, la mise en œuvre doit prendre en compte l'accessibilité non prévisible des fournisseurs, ainsi que les délais potentiels de la communication.

Cette thèse propose une plate-forme de services pour les terminaux mobiles, qui supporte l'exécution d'applications orientées services dans les MANETs discontinus. La plate-forme consiste en un intergiciel structuré en deux couches : une couche de communication et une couche de services. La couche de communication permet le découplage entre deux entités en termes d'interactions dans le temps, de comportement synchrone, et de connaissance mutuelle. Pour cette couche de communication (DoDWAN), un protocole opportuniste et basé contenu est utilisé. La couche de services (DiSWAN) permet le découplage entre les fournisseurs de services et les clients en termes d'interopérabilité et de contrat de service. Pour cette couche, des solutions de découverte et d'invocation de services sont proposées.

Les éléments du protocole de découverte sont la description et la publication du coté fournisseur, et la collecte et la sélection du coté client. La description inclut des propriétés fonctionnelles et non-fonctionnelles du service, ainsi que des propriétés contextuelles. La découverte est basée sur le modèle pair à pair, où un client ne collecte que les services intéressants. Les invocations sont basées contenu pour bénéficier de la réplication de services, dans le cas où plusieurs fournisseurs peuvent fournir le même service métier. Le client crée sa requête en connaissant un fournisseur déjà découvert, mais il publie cette requête afin qu'elle soit reçue par tous les fournisseurs compatibles. L'exploitation de la multiplicité des fournisseurs pouvant engendrer des communications superflus, plusieurs mécanismes de "guérison" du réseau ont été mis en place pour éliminer les requêtes et réponses d'invocation redondantes. Des simulations dans un MANET discontinu ont été effectuées. Les simulations montrent que la performance de la découverte étant directement liée à celle du protocole de communication et que l'utilisation des invocations basées contenu permet de meilleurs temps de réponses et une meilleure satisfaction du client. Les simulations montrent aussi que les mécanismes de "guérison" du réseau sont efficaces.

Abstract

Mobile ad hoc networks (MANETs) are spontaneously formed out of mobile devices that communicate thanks to short-range wireless communication capabilities (e.g. Wi-Fi, Bluetooth). In many ad hoc networks deployed in real conditions, mobile devices can exhibit highly dynamic behaviors of mobility and volatility. Because of their behavior and their distribution, the devices in such network environments form so-called "islands" whose topology evolves continuously, rather than a single fully connected network. In this work, I focus on this class of MANETs which I refer to as *disconnected* MANETs. Network-wide communication in disconnected MANETs is still a challenge, namely because routing techniques designed for fully connected MANETs cannot be applied. The *store-carry-and-forward* approach provides a solution. With this approach, a message can be stored temporarily on a node, in order to be forwarded later when circumstances permit. Mobility then becomes an advantage as it facilitates message propagation from one island to another.

The service-oriented computing (SOC) model seems suited for ad hoc environments because it emphasizes the decoupled nature of its entities. Effectively, the decoupling between a client entity and a provider entity becomes essential in mobile environments with a fluctuating availability of providers, and where end-to-end communications are not guaranteed. Still, in existing service-oriented systems, providers are usually supposed to be available and always reachable. Therefore, implementing distributed services for such networks still poses several challenges. Not only network-wide communication features must be provided, in spite of constant network fragmentation, but aspects such as the unpredictable reachability of the providers, or potential communication delays, must be taken into account at the service level.

I propose a service platform for mobile nodes that supports the execution of service-oriented applications in disconnected MANETs. The service platform is implemented as a middleware composed of two layers: a communication layer, and a service layer. The communication layer provides mechanisms to decouple two service entities in terms of temporaneous interaction, synchronous behavior, and mutual knowledge. For this layer (DoDWAN), I use an opportunistic and content-driven protocol. The service layer (DiSWAN) provides mechanisms that decouple service providers and clients in terms of interoperability and service contract. For this layer, I propose solutions for service discovery and invocation.

The elements that construct the discovery protocol are the description and advertisement at the provider side, and collection and selection at the client side. Description includes functional and non-functional service properties, as well as context properties. Discovery is based on a peer-to-peer architecture, where a client collects only interesting services. Content-based invocations are used to benefit from the service replication, in case a multitude of providers can be offering the same business service. A client creates a functional request according to a discovered provider, then publishes its content-based request message to the attention of all compatible providers. Profiting from this provider multiplicity can generate redundant messages, I propose network healing mechanisms in order to eliminate the redundant invocation requests and responses.

Using a prototype implementation of the middleware, I conducted simulations in a disconnected MANET environment. The performances of discovery is directly related to those of the underlying communication layer. Simulations confirm the benefits of content-based invocations in providing faster response times and better client satisfaction. Simulations also show a drastic reduction of redundant and leftover messages when applying the healing mechanisms.

n d'ordre : 223

Université de Bretagne Sud

Centre d'Enseignement et de Recherche Y. Coppens - rue Yves Mainguy - 56000 VANNES

Tél : + 33(0)2 97 01 70 70 Fax : + 33(0)2 97 01 70 70