p2pSOA: A MIDDLEWARE ARCHITECTURE TO ENABLE GROUP COLLABORATION THROUGH SERVICE COMPOSITION

A Dissertation Presented

by

Demetris Georgios Galatopoulllos

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

December, 2011
Abstract

Digital personal devices, such as Personal Computers and mobile phones, have become pervasive in our times. We more and more depend on their services for our daily tasks from sharing information to managing activities among groups of friends, family, or business peers. With the recent advances in “smart phone” hardware and software technologies it is also now possible to access social networking sites or share content through the Cloud from anywhere at anytime. Moreover, web servers can now be deployed on mobile devices and provide software services to remote users. These new capabilities have far reaching implications since they allow mobile devices to execute services and participate as “equal citizens” in service compositions that can offer enhanced user experience and functionality not only to their owners but also to whole groups (communities) of collaborating mobile peers. Personalized web servers also have the advantage of preserving user content privacy by keeping user data only local on private devices and offering it only to trusted peers.

We have designed, implemented and validated a novel middleware, called Peer-to-Peer Service Oriented Architecture (p2pSOA) which can run on both personal computers and mobile phones and enable the sharing, distributed discovery and execution of composite services among trusted peers. Specifically, p2pSOA offers naming and P2P addressing of private services in devices which reside in NATed networks directly i.e. without the need for using any service-level proxies or intermediaries. Users of PCs and mobile phones are authenticated into trusted peer groups (communities) that may collaborate by volunteering their services for service composition. The p2pSOA middleware supports the true Peer-to-Peer execution of distributed SOA applications which implement use scenarios combining private services running on such devices with services executing in the public open Web. By keeping the user’s sensitive content
on personal devices within the trusted group, it becomes possible using p2pSOA middleware to preserve user privacy.

P2pSOA also enables the seamless integration of the P2P service model with Cloud service provision, thus enabling service providers to offer services not only to individual users but also to communities of trusted peer. Furthermore, this is possible without the need for service providers to adopt an Application Programming Interface (API) specific to p2pSOA. Composite Service descriptions (applications) remain agnostic of the underlying p2pSOA middleware which handles all the service “plumbing” tasks.

We have implemented and tested a working p2pSOA prototype which adopts the Web Services execution model and utilizes scripting on mobile devices to allow the automatic generation of client stubs at runtime without the need for code compilation. P2pSOA supports REST-style services hosted on mobile devices and both SOAP-based and REST-style services on personal computers, regardless of their location, and it combines them with Internet services into composite services. The prototype utilizes Oracle’s JXTA as a secure P2P overlay network without modifying the software in any way. It achieves this by using adapters that can be substituted to create different p2pSOA middleware instances. Different types of transport and distributed service layers can be accommodated making p2pSOA agnostic to their runtime.

Finally we demonstrate and evaluate the capabilities of the middleware through several interesting collaborative computing use case scenarios.
Contents

List of Figures........................................................................................................................................ viii
List of Tables ......................................................................................................................................... xi
Acknowledgments .............................................................................................................................. xii
Publications ........................................................................................................................................ xv

Chapter 1 ..............................................................................................................................................1

   Introduction........................................................................................................................................ 1

   1.1. Research Motivation.................................................................................................................. 2

   1.2. Research Problem.................................................................................................................... 4

   1.3. The p2pSOA Vision ............................................................................................................... 7

   1.4. The p2pSOA Service Model................................................................................................... 10

   1.5. Contributions......................................................................................................................... 12

   1.6. Roadmap............................................................................................................................... 14

Chapter 2 ............................................................................................................................................16

   Related Work .................................................................................................................................. 16

   2.1. Social Networking, Clouds and the Online World............................................................... 17

   2.2. Peer-To-Peer Computing........................................................................................................ 18
4.1. The User Web Interface (UWI) ....................................................................................... 64
4.2. The Composite Service and the Stubs ............................................................................. 67
4.3. The standard p2pSOA ..................................................................................................... 68
4.4. The Mobile-p2pSOA ....................................................................................................... 69
4.5. Common Implementation................................................................................................ 70
4.6. The p2pSOA End-to-End Message Routing ................................................................. 71
4.7. The p2pSOA Runtime ..................................................................................................... 76
  4.7.1. Initialization Phase.................................................................................................... 76
  4.7.2. Discovery Phase Implementation ............................................................................. 79
  4.7.3. Cloud Discovery Implementation ............................................................................. 82
  4.7.4. Execution Phase Implementation.............................................................................. 82

Chapter 5 ......................................................................................................................................87
p2pSOA Evaluation ............................................................................................................... 87
5.1. Requirements for the new Middleware and the p2pSOA Architecture ......................... 87
5.2. Use Cases Realization ..................................................................................................... 88
  5.2.1. Snap-A-Picture: Control the Mobile Phone’s Camera Remotely ............................. 88
  5.2.2. Family Errands: Organize the Grocery Shopping Between Family Members ......... 92
  5.2.3. Cloud and Peer-to-Peer Overlay Integration: Discovery Services through the Cloud ................................................................................................................................. 96
  5.2.4. Book Search: Sharing of Book Lists Between Friends............................................. 99
List of Figures

Figure 1.1: The networked device ecosystem. Devices can be connected to heterogeneous networks, roam across network boundaries, or reside behind Firewall/NAT routers..................................................................................................................... 5

Figure 1.2: The p2pSOA vision.............................................................................................. 7

Figure 1.3: The conceptual static class diagram for the p2pSOA design......................... 9

Figure 2.1: The Web Services based SOA model using a broker for publishing and discovering the service descriptions. .............................................................. 22

Figure 3.1: The functional diagram for the Snap-A-Picture application: ......................... 44

Figure 3.2: The UML sequence diagram for the Snap-A-Picture application. .................... 45

Figure 3.3: Functional description of the Family Errands application. .............................. 46

Figure 3.4: The UML sequence diagram for the Family Errands application. ................... 47

Figure 3.5: The functional diagram for the integration of the Cloud in application. ........... 48

Figure 3.6: Use Case diagram for the proposed middleware architecture : ....................... 49

Figure 3.7: A layered architecture supporting Web Services composition and execution.. 51

Figure 3.8: The p2pSOA middleware layer incorporated in the general Web Services architecture. .......................................................................................................................... 52

Figure 3.9: The p2pSOA functional subsystems and their dependencies............................. 54

Figure 3.10: Discovery Phase for discovering a private Web Service. ............................... 57

Figure 3.11: Discovery Phase for discovering a public Web Service. ............................... 58
Figure 3.12: The Execution Phase of a composite service composed of public and private services. .......................................................................................................................... 61

Figure 4.1: The p2pSOA User Web Interface HTML page. ...................................................... 64

Figure 4.2: The User Web Interface module interactions during discovery and execution. 65

Figure 4.3: The Family Errands Composite Service and Stub module. ......................... 66

Figure 4.4: WSDL fragments for the Books SOAP service: (a) the original WSDL file and (b) the modifications performed by the p2pSOA end-to-end protocol. ........ 73

Figure 4.5: SOAP (top) and REST stub segments showing the runtime routing information. ....................................................................................................................... 74

Figure 4.6: The p2pSOA Initialization Phase; Steps with the same index can occur concurrently or in any order between them.............................................................. 76

Figure 4.7: The Discovery Phase implementation................................................................ 78

Figure 4.8: UML sequence diagram for the Discovery Phase............................................. 79

Figure 4.9: Activity diagram for the incorporation of Cloud services in the Discovery Phase......................................................................................................................... 81

Figure 4.10: The Execution Phase implementation............................................................. 83

Figure 4.11: UML sequence diagram for the Execution Phase........................................... 84

Figure 4.12: The activities during the execution of private and public services. .......... 85

Figure 5.1: Network diagram depicting the configuration of devices for Use Case A........ 90

Figure 5.2: The server-side phone (left) and the client-phone connected to two separate home networks............................................................................................................ 91

Figure 5.3: The server-side phone (left) and the client-side phone during an instance of a picture snap.................................................................................................................. 92
**Figure 5.4:** Network diagram depicting the configuration of devices for Use Scenario B... 94

**Figure 5.5:** The Google Map canvas on the client showing the closest grocery stores. (red markers). The orange circle pinpoints the location of the client............... 97

**Figure 5.6:** The phone closest to the store receives a notification that a grocery list is available for viewing (leftmost image). If the list is accepted then the map of the phone location and the location of the closest store are displayed on the phone..................................................................................................................... 98

**Figure 5.7:** Network topology for the BookSearch use case. ..................................................... 99

**Figure 5.8:** Network setup for benchmarking the standard p2pSOA reference implementation ................................................................. 101
List of Tables

Table 1.1:: Comparison of existing service composition middleware designs with the p2pSOA middleware solution.......................................................... 38

Table 5.1:: Comparison of p2pSOA with direct HTTP.................................................. 102

Table 5.2:: Metrics for the Book Search application .................................................... 103

Table 5.3:: Memory footprints of the standard p2pSOA .............................................. 104
Acknowledgments

The doctoral degree is a grand undertaking that requires a balanced focus on course work, research, and personal development. One of the most important aspects of this process is the proper mentoring and guidance. I was very fortunate to have a stellar committee that has helped me tremendously along the way, from the early stages of being a graduate student and the initial inception of the vision, to the final stages of the design implementation. First and foremost, I want to express my gratitude to my advisor Professor Elias Manolakos who has mentored me and guided me through this process and who has believed in my potential to complete the work at the highest level of quality. His strong work ethic and his dedication to academic research has helped me shape my own path in this field.

I want to also extend my gratitude to Dr. Dimitris Kalofonos, collaborator and friend, for his endless efforts during all phases of the project. His ideas, insights, and guidance helped us solidify the original ideas into a concrete product that now opens many doors for future work in this domain. And I want to extend my many thanks to Professor David Kaeli for his support throughout the PhD process, his valuable feedback on the thesis, and his encouragement to further extend this work in the future.

A great level of support also originated from a number of external resources. Many thanks go to Professor Panayiotis Takis Metaxas for his invaluable contribution of development mobile phones to the project. These phones are not easy to obtain and we were very fortunate to hold on
to them for such an extended period of time. Special thanks also Dr. George Kontopidis for making the time to review our vision and give us his expert feedback and industry perspective on our directions.

I am grateful for the strong support I received from the Department of Electrical and Computer Engineering throughout my years here at Northeastern University. My many thanks to the ECE Chairman, Dr Ali Abur, and to Professor Hanoch Lev-Ari and Professor Masoud Salehi for the strong support they have provided me that helped me fulfill my goal. I also want to thank Professor Amir Farhat for the support he provided to me throughout my teaching assignments, and more importantly, for his valuable advice and encouragement in all aspects of the academic process. Also, special thanks to Prof. Shafai and David Potter for their valuable support. I am grateful to the ECE and COE staff who have helped me with all of the necessary administrative matters. Special thanks to Faith Crisley, Sharon Heath, Linda Bonda, Ellen Zierk, Lisa O’Neil, and Steve Gibson for their continuous support on all aspects of my degree requirements. Finally, I wanted to express my gratitude to the system administrators of both the ECE and COE, and specifically Fred Dellaporta and Tom Papadopoulos, for all of their help with the computing resources that were necessary to complete my work.

I was very fortunate to have worked in the CDSP Lab, next to very bright and talented students, who became my very good friends throughout these years. Thanks to all of you for making my tenure in the lab worth every minute of it. Special thanks to Joan Pratt who has provided us with everything that was needed to make the lab a highly functional work environment. Many thanks also to a number of my friends who provided me with resources or support to complete the testing phases of my design. In particular I wanted to thank Fatemeh Fazel for “sacrificing” her personal mobile phone and her home network to my critical software
testing. And finally, thanks to all of the people in the various forums, who although they didn’t know me personally, they took time out of their busy schedules to give me feedback and discuss my issues with the various software packages that I integrated in my design.

My parents strongly believed in the value of higher education, worked hard, and made endless sacrifices to make sure that I receive such golden opportunities. I cannot express enough gratitude to my mother Electra and my late father Giorgos, for all of these sacrifices so that I can achieve my dreams. I extend my gratitude to my brother Themis and sister Diana for their support through the years.

Finally, this thesis is dedicated to my family, Lori, Giorgos-Alexander, and Sofia-Nicole. Without their love, support, and patience, none of this would have been possible. They made the journey worthwhile and taught me how to strive for the good.
Publications


Chapter 1

Introduction

Personal devices, such as mobile phones, digital music players and digital cameras are commonplace in our lives. Most of these devices are now manufactured with powerful CPUs and large RAM sizes making it possible to deploy complex software applications on them, including full-featured Web Servers. These advances have transformed the user experience and capabilities on mobile devices to a comparable form of what personal computers can offer. Moreover, the user now has more control of what can be customized on a mobile device and how the functionality can be extended through 3rd-party software applications. Therefore, with these devices fully network-aware, it becomes technically feasible to harvest and combine their services with services offered by personal computers and with services on publicly available Web Servers, and deliver to users, groupware and collaborative capabilities with an enhanced user experience. The consequence of such a model will be that users will then be able to access and combine services offered by devices of their peers, regardless of their location. For example, a group of friends who decide to collaborate by contributing pictures stored in their digital cameras could build custom albums, or a family relative overseas could share our home web camera. Furthermore peer services could be combined with known public services, such as Google or Yahoo Maps, and further expand the capabilities of existing applications.
example, a group of friends can share their locations so that the one who is closest to a store can pick up groceries for dinner.

So how can programming-agnostic non-expert users reap the benefits that distributed composite services have to offer without compromising privacy? How can devices that operate in private networks offer their services to trusted peers?

1.1. Research Motivation

Composing distributed heterogeneous services hosted on mobile devices and personal computers with services offered by Service Providers can offer users a powerful model of collaborative capabilities. Furthermore, these capabilities can extend the functionality of social networking engines by enabling communications through application functionalities. Traditionally, the Social Networking model has enabled users to communicate by sharing or accessing content on the Web. Augmenting such technologies with services on personal devices in a decentralized manner can have far reaching benefits for Users and Service Providers alike. For example, users can place content that they want to share online and then augment that with private content kept and shared across peer devices. Similarly, web technologies, such as the Cloud [1], can be utilized to facilitate runtime discovery of services that are otherwise not available on distributed peers. Using the Cloud, peers or Service Providers can recommend alternative services to fulfill the desired compositions.

Evaluating the example scenarios described at the beginning of the chapter we define a set of requirements for realizing these vision:

- The end-users should be able to seamlessly access and execute each other’s personal services even if they are not publicly available. Mobile devices and personal computers
are not always connected to public networks and this hinders the ability to reach their services directly.

- The end-users should be able to combine personal services with publicly available internet services, such as Google or Yahoo maps, into composite services with enhanced capabilities.

- The composite services should be executed in a distributed manner. The compositions can be launched from a single point on the client’s device and then the execution is disseminated to the involved parties. The results are returned to the client for final processing.

- The underlying infrastructure should preserve user content privacy. Social networks and the Cloud are web-based centralized technologies requiring that the users upload their content to their servers. User content privacy will enable the users to keep their content on their devices and volunteer access to them to trusted peers only.

In order to fulfill these requirements there is a need for a new middleware architecture that can mediate between clients seeking to compose services and the servers that host them. The concept of a middleware is generally defined as a set of services that lie between the application layer and the connectivity layer [2]. It hides the complexity details of the lower layers so that developers can implement portable and interoperable applications. Middleware architectures in the context of this domain provide a set of core services that hide the heterogeneity of devices and enable their services to be exposed at the application layer [3]. Examples of these services include service discovery, semantic matching, distributed execution, context-awareness, trust and security.
Therefore, the middleware architecture that can meet our set of requirements should be able to seamlessly offer the above capabilities to the stakeholders of the service compositions, namely the end-user and the Service Providers. These capabilities will allow the creation of ad-hoc Peer-to-Peer (P2P) communities that promote collaboration between trusted peers in a group through distributed service and content sharing while preserving privacy. Furthermore, it will enable mobile users to contribute services hosted on their devices with the community. With the same token, mobile devices can be used to request and compose services that the community offers while they are on-the-go anywhere anytime. Finally, service providers can advertise and offer value-added community services seamlessly through the Cloud. By seamlessly here we mean that the middleware will hide the location of the services involved in the composition from the client applications. This can promote the development of portable applications that are not bound to particular service implementations.

By following the principles of the Service Oriented Architecture paradigm [4] implemented through the Web Services model, this middleware can enable services to be discovered and composed over Peer-to-Peer overlay networks and the Cloud, offering to the peers of a trusted group a collaborative environment.

1.2. Research Problem

The ecosystem of distributed devices that can be involved in a service composition is shown in Figure 1.1. Nowadays, network-enabled devices can connect to home networks with routers that implement Network Address Translation (NAT) [5], they can be deployed behind enterprise firewalls, or they can roam within constrained operator’s networks. Mobile devices can continuously connect and disconnect from Wi-Fi networks while their users move from location
to location. Based on these observations, there are three classes of problems preventing the vision of composing public services with private services residing on mobile devices and personal computers to become a reality. First, most personal devices are not commonly connected to public networks but are rather connected to networks that implement NAT. NATs mechanisms assign IP addresses that are not valid in the open Web and access to these devices and their services is not possible through protocols that rely on direct IP addressing. Also, mobile devices introduce the issue of IP changes due to mobility. As nodes move across networks, their addresses change rendering any application that relies directly on those addresses useless. This general problem of *pervasive connectivity* entails dealing seamlessly with such firewall traversals.

**Figure 1.1 (1):** The networked device ecosystem. Devices can be connected to heterogeneous networks, roam across network boundaries, or reside behind Firewall/NAT routers.
and NAT issues and with changes to IP addresses due to the mobility of nodes. It is also concerned with the heterogeneity of the various platforms each device operates on. Finally it includes the issues of service-level naming and addressing: in order to maintain portability, composite services are developed using abstract naming conventions and rely on the underlying mechanisms to handle the naming and addressing of the services they invoke.

The second class of obstacles is the issue of security, privacy, and trust. Security is about controlling and managing access to resources over the network. Privacy, on the other hand, is concerned with the management of the amount of information a user is willing to share with others. On one hand, users are not always keen in exploiting the security features on their devices [6]. On the other hand, some of these devices have resource limitations, and executing complex security software on them is not ideal. In other cases, users are unaware of the privacy breaches that their devices may render. For example, the mobility of the user gives rise to issues such as location privacy. Finally, decentralized systems are usually large collection of heterogeneous and unfamiliar entities taking part in spontaneous collaborative computing and communications. For this reason, devices in such systems should be trusted by their peers before they can exchange information. In computing, this trust is defined as the measure with which a peer does what it is expected to do according to a set of contextual policies [6].

Taking all these requirements into consideration, we envisioned a middleware architecture that implements a service model to include the services offered by networked devices regardless of their location or implementation. This middleware, titled p2pSOA, deals seamlessly with the issues of pervasive connectivity and adopts 3rd party software in order to deal with security, trust and privacy.
1.3. The p2pSOA Vision

The high-level view of the p2pSOA vision that can handle the above requirements is depicted in Figure 1.2. Expert Developers and Application Vendors create services and their description documents. Service Providers make these services available to Users that in turn deploy them on their devices. The vision of p2pSOA is to mediate between trusted client devices that execute composite services and trusted server devices that host services, the services offered on the Cloud, and the services in the public domain, and to offer discovery, end-to-end routing and execution of composite services. This capability will enable the creation of ad-hoc Peer-to-Peer (P2P) Communities based on trusted groups where peers are authenticated by the P2P overlay for sharing and accessing services. The middleware executing on all of these devices will further extend the compositions through an infrastructure that interconnects peer services with services on public servers. Adding to this capability, Service Providers (SP) can offer value-added
services and their compositions through discovery mechanisms that utilize Cloud technologies. Both the Cloud spaces that are accessed through user authentication methods and the user devices that are connected to private networks are considered private spaces. The web servers with public Internet addresses are considered part of public spaces. Public services are addressable through public IP addresses in the open Web. Public Services are offered by the SP.

This model enables peer collaboration by enabling members of a trusted group to contribute services voluntarily, directly or through the Cloud (e.g. Dropbox), so that other group members can combine them with their own services into service compositions that can be executed in a distributed fashion. This groupware capability enables the members of the group to get organized towards common goals. One characteristic example would be to help a group of elderly neighbors organize trips to common points of interest that are close in proximity to their residences.

Currently, a number of middleware designs exist that address a set of issues related to composing services across heterogeneous devices regardless of their location. Some of them have dealt with the issues of pervasive connectivity by presenting a solution to composing services behind NAT networks with publicly available services. Other designs explored the provision of services on mobile devices and their inclusion in larger scale service compositions. A number of designs have utilized the Cloud to deliver services to residential networks. A comprehensive review of service composition middleware is provided in Chapter 2.

Although most of these designs offer solutions for a Peer-to-Peer SOA middleware, these solutions only partially satisfy the requirements defined above. There is not a middleware solution today that solves all of the problems described above. Namely not a single design can (i)
Figure 1.3 (3): The conceptual static class diagram for the p2pSOA design.

solve the issue of pervasive connectivity using peer-to-peer technologies in order to combine private and public services regardless of their location, (ii) incorporate mobile services in the...
composition, (iii) utilize the Cloud to discover value-added services, and (iv) execute the compositions in a distributed privacy-preserving manner. The p2pSOA design offers an integrated unique solution meeting simultaneously all of the above requirements.

1.4. The p2pSOA Service Model

Based on vision defined in the previous section, we define the conceptual static class diagram in Figure 1.3 that describes the p2pSOA service model. In this view, the Group is a service for enabling distributed peers to form a virtual secure network. A peer in p2pSOA is a networked entity that has the capability to host and provide services. Each peer can use the Group’s join mechanism to become a Member of the Group. The Group offers publish and discover services to authenticated Members that enable them to publish their resources and discover the resources of other Members of the Group. We define two specializations of the Member type: Personal Computer and Mobile Device. Both these types can operate as providers and as requesters/consumers of services. Members of the trusted Group host services and they are responsible for exposing their interfaces for consumption.

Services are software modules based on the Web Services model and their capabilities are defined in interfaces described with Web Services Description Language (WSDL) documents [7]. User devices can be both a producer and a consumer of such services. Hereafter in this thesis, the terms “server-side” and “client-side” are used only to denote which peer is the service producer or the service consumer in a particular use case scenario. This is because peers in the p2pSOA context can both be clients and servers. Services can be atomic or composite. Atomic services are the services that offer a single cohesive functionality that can be combined with other atomic services to offer more complex operations. Atomic services and their WSDL
documents are generally developed by expert programmers. In the context of p2pSOA we define *elementary services* as the atomic services offered by any device participating in the composition. We use the term elementary or atomic interchangeably in this document.

From the point of view of network accessibility, services in p2pSOA are classified in two specializations, *private* or *public*. Users and their devices can roam both in public Wi-Fi networks and also in private networks (e.g. home networks) with routers that implement NAT. Privately-owned personal devices that connect to such NAT networks, or operate behind Firewalls, do not have the infrastructure to expose services deployed on them globally through public IP addresses. We define such services as private. Public services are available in the public domain, e.g. Web Services that are accessible over the Internet.

A *composite service* (CS) is an aggregation of Services. A Composite Service can include one or many Services. Both types of Members in the p2pSOA can execute a Composite Service when acting as clients to a service composition operation. Since one of the primary goals of the p2pSOA middleware is to enable the execution of composite services, we handle a CS as a client module. The CS is developed with calls to service names. These service names are generic abstract names that are publicly known and generally describe the functionality of the service. For example, a service with the abstract name Map refers to a service that draws a map of a location. The abstract name is not bound to a particular Map service implementation. The abstract service names are thus an indirection pointing to the communication stubs that are generated by p2pSOA. They achieve a separation of the composite service code from the communications code. The CS can then be developed portable to the p2pSOA architecture by expert programmers and offered to the user community through a *Service Provider* (SP). A composite service in p2pSOA is comprised of, a composite service client module that executes
the business logic, and a textually defined workflow description, that describes which atomic services are needed in the composition.

The stakeholders to the atomic services and the service compositions are, the Service Providers that offer them, and the users of personal computers and mobile devices that subscribe to the SPs and consumer these services. Users in this model should have access to the Internet and to Web-based technologies such as the Cloud. Service Providers offer the atomic services over the Internet and the users select, download and deploy them on Web Servers on their devices. The corresponding WSDL documents are entered in databases stored on the devices.

A service composition operation is composed of two actions: Discovery and Execution. Discovery is delegated to the p2pSOA entity. The p2pSOA entity in the schematic represents a middleware layer that mediates between the peers and the distributed services. During discovery, a set of Services are identified across peers based on a textual description of the composition. For services not found on peers, the p2pSOA accesses the Cloud for discovering public services recommended by the peer. During Execution, the invocation to Private Services is mediated through the p2pSOA whereas the invocation to Public Services is transported directly to the Services through standard Web tools, such as a browser. Finally, the Cloud entity in the diagram is a representation of a Cloud application. The Cloud is used by Peers or the SP to publish location information of Public Services that are discovered by the p2pSOA during service discovery.

1.5. Contributions

In this dissertation work we have defined and implemented the p2pSOA middleware architecture that overcomes the obstacles and offer the capabilities and solutions described in the previous
sections. It facilitates the sharing and distributed execution among trusted peers of composite services consisting of private and public services using both mobile devices and personal computers.

The contributions of this work are:

- The design of the p2pSOA middleware architecture which combines the benefits of the P2P computing with the heterogeneity aspect of the SOA paradigm and offers:
  - naming and addressing of devices that reside in networks that implement NAT without the need of service-level proxies or intermediaries.
  - the participation of mobile devices in communities of users that collaborate through service composition.
  - the sharing of services and content between trusted peers through the distributed execution in a true P2P fashion of composite services which combine private and public services.
  - the integration of the P2P model with the Cloud model enabling service providers to offer community services to trusted groups of peers through the Cloud.
  - privacy of user content by keeping the user information only on personal devices. Trust is accomplished by utilizing the concept of trusted group in which peers can trust the services offered by authenticated members of the group.

- The development of a working p2pSOA prototype that:
adopts the Web Services execution model and utilizes scripting on mobile devices to allow the automatic generation of client stubs at runtime without the need for compilation.

- supports services hosted on both mobile devices and personal computers.
- combines private (device-level) and public (internet) services into composite services
- utilizes JXTA as a secure P2P overlay.
- integrates Cloud capabilities for discovering services.
- handles both SOAP-based Web Services and REST-style Web Services

- We did not modify in any way the 3rd party software packages and libraries that were incorporated in the implementation of the p2pSOA middleware. Instead we designed interfaces (adapters) that can be substituted to create different p2pSOA instances. Different types of transport and distributed service layers can be accommodated making p2pSOA agnostic to their runtime.

1.6. Roadmap

The rest of the thesis is organized as follows:

Chapter 2 introduces the Social Networking and Cloud technologies and the issues of privacy. It then describes the Peer-to-Peer and SOA technologies necessary to design and develop middleware for service composition that overcomes the issues of pervasive connectivity. Finally it presents a survey of the work related to our design. The survey reveals the lack of an integrated middleware architecture solution to solve the research problem defined in Chapter 1.
In Chapter 3 we use the results of the related work survey in Chapter 2 in conjunction with a set of motivating Use Scenarios in order to define the requirements and specify the p2pSOA middleware architecture. In the remainder of the chapter we outline in detail the p2pSOA middleware components and how they contribute in the distributed service Discovery and Execution phases.

Chapter 4 describes the systemic and functional view of p2pSOA and the implementation of the p2pSOA proof-of-concept prototype. The prototype was implemented to execute on personal computers and then adapted to also operate on mobile devices. We then describe in detail the sequence of events that occur during the Discovery and the Execution phases of the implemented prototype.

Chapter 5 outlines the implementation of the motivating applications described in Chapter 3. We specify four major applications and we outline the corresponding Use Scenarios that we used to implement them. We analyze the results of the Use Scenarios from the user experience point of view for the first three applications. The fourth applications was implemented to profile the performance of the software implementation. We present the numeric results of this analysis in detail.

In Chapter 6 we summarize the contributions of the thesis and we outline the conclusions based on our experience with its implementation. The innovations of this work points to a number of interesting future directions in the area of social networking and smart homes. We outline these directions in detail.
Chapter 2

Related Work

In this chapter we present a focused analysis of the existing work on middleware that enable service composition and we describe the technologies that are employed for their implementation. The first section describes the concepts of Social Networking and the Cloud that have transformed the Web to a user-friendly tool. The explosion in popularity of these technologies, however, has also brought about concerns in the area of user-content privacy. This has motivated our solution to employ peer-to-peer technologies as the messaging transport technology between peers. We describe the important concepts of peer-to-peer computing, how they help preserve privacy of user data, and how they deal with issues of pervasive connectivity.

In order to deal with heterogeneity across distributed devices, service composition middleware layers are designed with architectural styles that enable platform independence. The most prominent such style is the Service-Oriented Architecture (SOA) [4]. We describe the SOA model and one of its most popular implementations, the Web Services. For the rest of the chapter, we present middleware designs with solutions to the service composition issues that were described in the previous chapter and we compare them with the p2pSOA solution.
2.1. Social Networking, Clouds and the Online World

In the past few years, the Web has been transformed by a number of technologies from a black box into a user-friendly tool. One such technology is Social Networking (SN) [8]. As the name suggests, SN is a means of “connecting” with others by creating a virtual network through a set of web-based services. SN sites enable users to create and share a profile, post and share lists of friends, and create trusted groups. SN has brought about a revolution in how we communicate and collaborate with each other through technology. Facebook and Google have dominated this technology space bringing to users the necessary online tools to post and exchange content in a seamless manner. With the advent of the Smart Phones, the SN model was extended to mobile devices. Third party mobile social applications that enable users to access SN sites and continue to update and share content on the go have emerged making communications and content sharing through SN ubiquitous and as popular as ever.

Adding to this popularity, both Google with Google+ [9] and Nokia with Pulse [10] have recently launched social services and applications that enable users to create groups from their contact list and share with group members their context and information. A similar mechanism exists in Facebook where a user can create a private space, invite friends to join it, and then privately share information with them.

Another technology, that has helped shape the online world and how we perceive and use it to collaborate, is the Cloud. Cloud computing is a technology that offers software and resources as a service to individual users and organizations [1]. It extends the traditional model of running software and storing data on the personal computer (PC) by offering analogous capabilities online and over standard communication protocols. Although not a new field, Cloud computing has emerged as an elegant progression to virtualization of software and utility computing.
Storage Cloud services, such as the Dropbox [11], allow users to seamlessly transfer files between their desktop to online servers and be able to access and share that information anywhere anytime even from mobile devices. The Dropbox application appears as a desktop icon and the data can be dragged and dropped into it.

There is, however, an aspect to online collaboration that as been highly criticized [12]. SN and the Cloud are web technologies that inherently possess the issue of privacy and data security. Companies offering Cloud capabilities set forth comprehensive mechanisms with privacy policies and security measures for protecting personal information. However, personal data remains an extremely sensitive issue for many users particularly the non-technical users who are still reluctant to augment their personal devices and content with the Web. Therefore, solutions that can offer the capability to collaborate while keeping the user data only on trusted devices are appealing. The online technologies can then be used as extensions to such mechanisms offering additional functionality, such as publishing or discovering of services. Therefore service composition involving content sharing across trusted peers now points to solutions utilizing available peer-to-peer technologies.

### 2.2. Peer-To-Peer Computing

Peer-to-Peer (P2P) computing is an application-layer programming paradigm that utilizes Peer-to-Peer network overlays to achieve application to application multicasting [13]. P2P network overlays are networks residing over the Internet Protocol (IP) and are typically based on a layered network architecture that abstracts the details of network communications protocols used to connect distributed nodes, the management of such nodes, and the services that they offer. Unlike the traditional client-server model, overlay networks enable a two-way resource sharing
by allowing each peer to act as both a server and a client [14]. As articulated in [14]: “Peer-to-peer overlay systems go beyond services offered by client-server systems by having symmetry in roles where a client may also be a server. It allows access to its resources by other systems and supports resource sharing, which requires fault-tolerance, self-organization, and massive scalability properties. Unlike Grid systems, P2P overlay networks do not arise from the collaboration between established and connected groups of systems and without a more reliable set of resources to share”. Nodes belonging to an overlay network can share resources and services in a decentralized manner without the need for centralized management. This property is critical in preserving content privacy since the information shared in such networks is moved between nodes in a P2P group but never reaches any 3rd party server nodes.

Although the Service-Oriented Architecture model offers interoperability and platform independence, the services involved in SOA implementations are generally assumed to be publicly accessible. However, services that reside on hosts behind NAT cannot participate in distributed computations. Although the IPv6 is becoming a reality, the majority of networks, including the home network, still operate under IPv4.[15]. This means that the personal routers utilized in such networks implement Network Address Translations (NATs) in order to deal with the scarcity of public IP addresses [16]. NAT is a technique applied at the boundary of a private network in order to enable hosts inside the network to access the public domain without requiring each one of them to have its own public IP address. The router of the network assigns IP addresses to networked devices in the local network which do not have meaning in the public domain. Therefore these devices are not addressable by anything outside the local network. To utilize private services from distinct private networks in compositions there is a need to allow distributed SOA components to traverse NATs and firewalls in a seamless manner. P2P overlays
can solve the problem of composing services of devices that reside in restricted private spaces by employing a number of techniques for NAT traversals.

There are a number of different NAT implementations, NAT applications and traversal solutions, as described in [5] and [17]. Although not a single general solution exists today that can deal with all NAT implementations, a robust method that deals with most common NAT types is called relaying [18]. In this method a host, the relay, is residing in the public domain and it is responsible for relaying (routing) messages between hosts that reside in private networks. Since none of the hosts behind a NAT are visible from the outside world, they are made aware of the relay peer, connect to it, and route all outbound traffic through it. This relaying method can be achieved through P2P overlay networks. One example of relaying through P2P overlays is the JXTA P2P technology [19]. JXTA utilizes a specialized Relay peer that routes messages between peers residing at the edges of the network that are otherwise unreachable.

Three designs have addressed the NAT traversal issue without using peer-to-peer technologies. These are the WS-Dispatcher (WSD) [20], the TARGET two-way SOAP router[21], and the IBM Web Service Gateway (WSG) [22]. However, all these projects either utilize intermediary proxies or centralized service registries. On one hand, the problem with intermediaries at the service-level is that they impose modifications to the service implementation. On the other hand, centralized registries are not fault-tolerant since a server failure may disable the system. Therefore there is a need for a distributed solution that also solves the problem of pervasive service connectivity without imposing any modifications to the service implementations. In order to accomplish this, a potential layered architecture can be designed to help separate the concerns of the transport operations from the services functionality.
Then a P2P overlay can be employed to solve the problem of dealing with NAT leaving the service implementation intact.

### 2.3. Service-Oriented Architectures and Web Services

The next concern in dealing with distributed services is heterogeneity. In order to discover and compose heterogeneous services there is a need for software architectures and standards that promote platform independence and protocol compatibility. One type of such architectures that has received a lot of attention in the past few years is the Service-Oriented Architecture (SOA) [4]-[23]. There are a variety of definitions as to what a SOA really is. In fact, the SOA definition varies depending on the stakeholder, for example, executives, IT managers, or developers [24].

The World Wide Web (W3C) consortium [25] defines SOA as a distributed systems architecture. The central concept in an SOA is the service, an abstract view of the entities involved in distributed computations. Similarly, the Organization for the Advancement of Structured Information Standards (OASIS) [26] defines SOA as a set of practices, principles and patterns related to service-aware, enterprise-level, distributed computing. In their seminal paper [4], Papazoglou, Traverso, Dustdar, and Leymann, define SOA as a methodology for building software that can deliver services to other services or end-users. As described in this paper, this methodology realized with well-established standards can enable organizations to provision, reuse and compose services on a dynamic infrastructure. As all of these definitions emphasize, SOA is not bound to a protocol or a platform, but it is rather a high-level software architectural approach for integrating distributed tasks in loosely coupled manner. According to [27], a Software Architecture is defined as an abstraction of the elements that comprise a system at a
particular phase of its run-time operation. Consequently, a software architectural style is imposed on a system in order to structure its components into a software architecture [28]-[29].

The concept of a service is central to the SOA. A service is an abstract entity that when realized by a software agent can offer a set of tasks to a requesting agent. In SOA the service is modeled along two concerns: the implementation and the interface. This separation enables interoperability since the service consumer will not be bound to a particular set of service implementations. The client applications are developed based on the contract (interface) and remain portable through service implementation upgrades.

The most commonly used model of SOA is the broker-based. As shown in the interaction diagram of Figure 2.1, resources advertise their software capabilities as public interfaces in a registry using service description documents. These documents are created using the machine processable Extensible Markup Language (XML) [30]. Services can be described in a syntactic or semantic manner. As the name suggests, the former method describes services as string-based
names. However, this can lead to naming conflicts and ambiguities. This issue led to the advent of the semantic descriptions concept. These descriptions are based on the Semantic Web. The Semantic Web (SW) was envisioned as an extension of the current Web where data is given a well-defined description so that it can be reused and processed automatically by standardized technologies with minimal human intervention [31]-[32].

In order to semantically describe services, the World Wide Web Consortium (W3C) defined the Web Ontology Language for Services (OWL-S) [33]. In general terms, an ontology is “a finite list of terms in a controlled vocabulary” that specifies a conceptualization [34]. This notion was adopted in the Web domain and a set of languages (Web Ontology Language (OWL)) were derived to enable applications to process information that otherwise would require human interpretation. The OWL-S is one such language that extends OWL with semantic annotations. It is composed of the service profile for advertising and discovering services, the process model for describing the service's operation, and grounding that refers to the message communications with the service. The ultimate goal of the OWL-S is to automate the service discovery, composition, execution, and monitoring i.e. to provide machine-understandable semantic descriptions to services so that applications can discover and compose them in an automatic manner.

Another popular semantic description specification with the same objectives is the Web Service Modeling Ontology, WSMO [35]. The two methods differ in a number of aspects [36]. OWL-S is a more mature technology than WSMO in describing the process and grounding models. On the other hand, WSMO provides the orchestration of Web Services and a more natural and intuitive method to describe the user’s requests.

Semantics have been successfully utilized for discovering and dynamically composing Web Services. Semantics are essential both in the service discovery as well as in the composition
phase in order to provide exact service matching and to enable the automation of the composition process. During discovery, the appropriate services that can fulfill the user’s task are semantically matched on the fly. Subsequently, specialized algorithms are used to generate the resulting composition.

2.3.1. Service Discovery

Service discovery and matching is a well-studied field and it is essentially a mechanism of identifying services that can satisfy a set of criteria defined by the user in a client application [37]. Clients query the registry and retrieve handles to these services that they can subsequently be used to invoke their methods. As mentioned earlier, simple discovery methods use a string-based matching approach, but with the advent of the Semantic Web it became clear that the service discovery and matching mechanisms can be semantically enhanced. Semantically enhanced services descriptions include annotations with meta-data that provide an information-rich description of their capabilities. The DAIDALOS project [38] utilizes semantic discovery in one of the service filters, a module that imposes criteria to service advertisements. This filter deals with semantically described services using ontologies. Services that match the criteria are accepted and the rest are discarded. The COCOA middleware [39] utilizes semantics to allow the user to discover and execute on the fly a task described on a networked device.

Service descriptions are typically advertised in online registries. Querying a directory-style registry to find services is one method of discovering services. For example, the UDDI-based (Universal Description Discovery and Integration) registry [40] is a centralized directory that stores Web Services described in WSDL [41]. The queries to the registry are sent as XML
documents inside messages based on the W3C standard Simple Object Access Protocol (SOAP) [42]. We will describe these technologies at a later section dedicated to Web Services.

Using centralized directories or registries to discover services presents the issues of single point failure and performance bottlenecks. Alternatives to the centralized schemes is using decentralized mechanisms. However, even after solving these issues, decentralized discovery mechanisms still need to deal with the issues of network and middleware heterogeneity of the devices involved in SOAs. For example, some devices may be using a different type of middleware that supports a different service discovery protocol than others in the same pervasive environment. One general solution for dealing with heterogeneity makes use of the concept of plug-ins. Plug-ins are adapters that allow the interchanging of various Service Discovery Protocols (SDPs). For example, the DAIDALOS design described earlier, deals with multiple discovery protocols through the Pluggable Discovery Module (PDM). The PDM is responsible for mapping a standard discovery request to a particular discovery protocol. Furthermore, DAIDALOS offers a four-phase filtering process of discovery results that can be further personalized by the user either manually or through an automatic integration. Similarly, the Multi-Protocols Service Discovery and Access middleware (MUSDAC) platform [43] makes use of specialized plug-ins that can interface with particular SDPs and retrieve from them service information. MUSDAC itself is registered with all these SDPs as a service in the local network and clients access it using their own SDP. As a request arrives at the MUSDAC service, it gets forwarded to the local network or remote networks for finding a suitable service.

The Wings project [44] follows a similar approach in order to support the discovery of pervasive resources over heterogeneous networks. It is based on the concept of plug-ins that is borrowed from a dynamic hierarchical component model called Compor [45], through which
features such as discovering or providing services can be introduced on constrained mobile devices. By interchanging these plug-ins, applications executing on the middleware can operate over different peer-to-peer protocols such as UPnP [46] and JXTA and utilize them to discover new services. Finally, the authors in [47] also present a layered middleware architecture for enabling SOA applications to discover heterogeneous device services. The Drivers layer, the lowest layer in their architecture, utilizes a unified language to describe the functional properties of the services executing on the device. The Bridges layer is responsible for interfacing standard service technologies, such as UPnP, Web Services, and Apache River Project (formerly Sun Microsystems Jini Project) [48]. The middleware instantiates the same service multiple times for each service technology space. By doing so, client applications that target a particular service technology will be given access to the instantiation of the service that belongs to that technology.

2.3.2. Service Composition

One of the major aspects of SOA is its support for service composition; the process of combining multiple distributed elementary services into an order of execution that yields a more complex service (composite service) that implements a user task. Elementary services are offered by parties free of charge or through subscription to clients that can compose them and reuse them to meet their requirements. In a more refined form, a group of users or companies can share services between them to form capabilities that can be shared within the group. Service discovery and composition are an integral part of the user-defined goal-matching process in SOA.

Service composition can be achieved statically or dynamically [49]-[50]. Static composition is based on services available at compile-time; dynamic composition on the other hand occurs at
runtime. The former method requires that services are executed based on a workflow before the execution occurs. A workflow defines a set of rules with which a (business) process can be automated and executed through the distributed sharing of artifacts [51]. In the dynamic composition method the compositions are satisfied at runtime with the help of specialized middleware that match service definitions to service implementations.

Simple composition matching techniques attempt to syntactically match method signatures. A method signature is the string representing the name and parameter list of a service method. More precise methods in identifying matching services are through semantic discovery and matching. Semantic service composition is discussed in [52]. This design is based on a goal-driven dynamic composition approach. A five-layer task-oriented service representation model allows the user to define the goals of the composition at the highest level. A resulting customized execution script invokes concrete services. The design in [53] is a framework for mobile ad-hoc networks and offers semantic composition of mobile resources. The user requirements are submitted to the system in a subset of the OWL-DL language [54] and both full and partial matches are allowed. Finally in [55], a novel semantics-based dynamic service-oriented composition architecture called Component Service Model with Semantics (CoSMoS), is presented. The architecture allows the user to intuitively request an application (composite service). It semantically discovers and composes the appropriate services that match the tasks in the requested composition using the Semantic Graph-based Service Composition (SeGSeC) module. Subsequently, the SeGSeC generates an executable workflow of the composed service. CoSMoS also includes the Component Runtime Environment (CoRE) middleware layer that supports various component technologies such as Web Services, CORBA [56], UPnP, and Apache River. CoSMoS does not compose services based on templates but rather uses directly
the semantics of components. This enables the rapid development of new applications without the need to generate templates for them each time.

In summary, the discovery and composition capabilities offered by SOA can contribute to the vision of incorporating devices in compositions. The personal devices in the market currently that can be weaved together to naturally and transparently serve the end-user’s application (task) are highly heterogeneous. Promoting software applications on them based on the SOA programming paradigm provides a uniform platform for the developer of applications.

2.3.3. Web Services

A popular implementation of SOA is the Web Services (WS) [57]-[58]. WS technologies are frequently synonymous with SOA even though there are classes of applications for which implementing SOAs with WS is not appropriate. A Web Service exposes the capabilities of a software entity or a device through WS standards and protocols. One key aspect to WS implementation is the adoption of the proxy pattern through stubs. By automatically generating the stubs a middleware can enable the execution of composite services on the fly.

Web Services and their properties are described in a Web Services Description Language (WSDL) document [7] typically stored at the host of the service. A WSDL is an XML document that describes services as network endpoints that are invoked to operate on messages that contain either document-oriented or procedure-oriented information. A mapping between the WSDL documents and the abstract names of the services they describe is also stored at the server. In the p2pSOA, the same mapping used at the peer level also exists on peer Cloud spaces (e.g. Dropbox) so that WSDL documents of services can be identified and discovered. Moreover, service descriptions may also be distributed by 3rd parties (service providers) through the Cloud.
The p2pSOA middleware supports WSDL 2.0 [7] that can describe both SOAP [42] and REST-style [28] services.

There are two predominant design methods for developing and deploying Web Services: SOAP and REST. The SOAP technology is based on the Simple Object Access Protocol specification defined by W3C [42]. It is an XML-based protocol that enables applications to exchange information over the HyperText Transfer Protocol (HTTP) [59] in an interoperable manner.

The second method is an architectural style called REpresentational State Transfer (REST). REST views the web as a collection of resources that are generally represented and described by documents. These documents are addressed through Uniform Resource Identifiers (URI) [60] and accessed by the client using explicit HTTP methods (verbs): GET, POST, UPDATE, PUT, DELETE. Although REST services follow the Resource-Oriented Architecture model [61], a model different than the SOA model, they can be used to implement SOAs.

Although a number of web servers have been implemented to support SOAP services on mobile devices as described in [62], REST-style services remain a more favorable solution since they don’t require any parsers for generating stubs and thus offer lighter solutions for such devices [63].

2.3.4. Service-Oriented Middleware and Peer-to-Peer Technologies

As described in [64], the future of the Internet will impose a number of challenges to the SOA architecture layers and in particular to the SOA middleware layer. These challenges will require that the SOA middleware deal with scalability, interoperability, mobility and awareness. These requirements will demand transformations to service description and discovery technologies. As the size of the Internet increases, there is a need for scalable discovery mechanisms that also deal
effectively with heterogeneity. Moreover, service discovery protocols should take into consideration the security, trust and privacy of user devices and data. The same factors apply to Service Access and Composition techniques.

Laying SOA-based middleware solutions on top of P2P networks can provide mechanisms that handle pervasive connectivity while dealing with heterogeneity and user-content privacy. Although such an integration will not solve all of the concerns for the Future Internet, we believe that it will enable future SOA architectures to deal effectively with service provision and their access on personal computers and mobile device, and with public and private services discovery and composition.

2.4. Related Work Survey

We present here a thorough survey of existing work in middleware solutions for service composition. We discuss and compare/contrast the designs based on the following criteria:

- Composition of private and public services: This category focuses on the composition of services hosted on devices behind NAT/Firewalls with services that are publicly available on the Web.

- Mobile Web Services composition: This category describes the involvement of services residing on mobile devices, private or public, in the service composition.
o Pervasive connectivity: This category refers to the capability of dealing seamlessly with devices residing behind NAT/Firewall, IP address changes for mobile devices, addressing and naming at the service-level, and dealing with heterogeneous platforms.

o Preserving privacy: With this category we investigate whether a particular design protects the user’s privacy by keeping the target data off public servers.

o Utilizing the Cloud for discovery of services: This axis refers to the integration of Cloud technologies with mobile middleware solutions to provide services through online directories. Such a service is the capability to discover services through the Cloud.

We begin with one of the early designs in service composition using mobile services as described in [65]. This work defines a set of protocols for composing distributed mobile services. Peers connected through ad-hoc networking protocols, offer and equally consume services offered by other peers. A layered architecture is defined that encapsulates the discovery of services, the service composition and execution, and the arbitration of a composition manager (CM). Any device in the ad-hoc network can be a CM and the best fit for a particular composition is selected by the composite request source node. Subsequently the CM carries out the composition execution. Although this design enables the composition of services that are distributed across mobile nodes, it does not deal with NAT and Firewall traversal, and it requires that the CM is aware of the address of each device. In addition, since it is only concerned with the composition of services offered by mobile devices, it does not have a provision for composition of private services with public services.
Similarly, the Mobile Service Composition API (MoSCA) design in [66], defines a middleware for developing and executing distributed mobile service compositions with high reliability. It achieves this through mechanisms that select the best available service providers during runtime. Programmers utilize a system-level MoSCA service to instantiate a locally visible MoSCA service object. With calls to this object, the service requests are bound to service providers that have high availability of offering the corresponding services. Although MoSCA’s application support model is effective, it does not deal with the pervasive connectivity across mobile devices. It assumes that connectivity is possible across all devices.

The design in [67], and its evolution into a decentralized SOA architecture [68]-[69], focuses on composing services across mobile devices using a peer-to-peer SOA. It utilizes a Distributed Hash Table (DHT) [70], a workflow engine, and a client-side programming model, to discover and transparently invoke services that reside on P2P overlays nodes with DHCP-assigned IP addresses ([71]) as well as behind NAT networks. The service providers utilize a gradient topology overlay to find a service registry, register a service with it, generate a service proxy and upload it to the DHT overlay. The clients find a service registry through the gradient overlay, retrieve semantic descriptions of services that match the composition criteria and then download the proxy from the DHT. The proxy is used to transfer service configuration information to the client. Similar to p2pSOA, the design deals seamlessly with pervasive connectivity. However, it requires that the service proxies and the service registry be replicated across a number of peers. In p2pSOA, each peer stores the WSDL files corresponding to the services deployed at that peer. This localizes the maintenance of valid WSDLs in the local database. If the peer leaves the group its WSDL file will no longer be available when the Discovery Phase takes place and it will not be included in the composition. This removes the
need for handling stale WSDLs in the registry. As mentioned, the [67] project utilizes a Service
Proxy that has the service information encapsulated in it. When the provider is contacted it
returns that proxy to the calling client. The service clients need to utilize an API in order to
download and utilize these proxies. This design handles binding and port agnostic issues through
the Abstract Protocol Adapter as described above. The APA is configured at runtime by the
Service Proxy to handle different service protocols. However, it does not dynamically generate
the code for calling these services but rather uses the Service Proxy which is a compiled Java
object. The service proxy is a Java object that needs to be generated, compiled and deployed for
a particular host. This limits this design from being deployed on mobile devices that lack
compilation tools. Also it eliminates the potential of migrating the service to another edge node
since the service proxy will need to be recompiled to include the new peer location’s
information. In p2pSOA the deployed services are not modified in any way. Instead, outgoing
requests are intercepted and modified on the fly in order to be routed to the appropriate peer. In
addition, client stubs are scripts that do not need compilation and are generated automatically at
runtime locally at the client. Finally this design does not utilize the Cloud for discovering
publicly available services and including them in the composition.

The design described in [72] defines a layered architecture that enables collaborative
composition of services across mobile devices. To remain relevant to current trends, the
architecture achieves this by using standard Web techniques, protocols and tools. With this
design, mobile devices interact with each other in a peer-to-peer manner offering and consuming
services through REST-style service invocations. The design is not primarily concerned with the
communications and addressing aspect of peer connectivity and instead it requires standard
HTTP enablement with public IP addresses for each device. It does, however deal with IP
address changes due to mobility. It achieves this through the Nokia Mobile Web Server connectivity solution. Since data is shared within personal groups, user-content privacy is preserved. Finally it does not utilize Cloud technologies for discovery.

The CoSP2P design [73] defines a service-oriented component model for the composition of services across mobile devices. It is built on top of the UM-RTCOM component model, a lightweight model that offers platform independent components for requesting and providing services and communicating with other components. The CoSP2P layer is laid over the UM-RTCOM and it utilizes the concept of the port as the access point to peer services. The port encapsulates the commands and events that others peers may invoke on a particular peer’s service. CoSP2P defines, but also imposes, a comprehensive programming model that is language and platform independent for developing services. Although it targets mobile services, it does not address the issues of pervasive connectivity or combining private services with public ones. Similarly, the MicroMAIS platform [74] supports the execution of atomic and composite Web Services across mobile devices. It is composed of the mobile Application Server (mAS) and the μ-BPEL engine. Both of these components were developed using Web standards in order to promote interoperability. This design does not directly address the issues of pervasive connectivity but it does contribute to preserving user privacy by composing services hosted on mobile devices.

A mechanism for accessing private services from the open Web is described in [75]. This design defines an architecture where services hosted in the private home Local Area Networks can be exposed to trusted peers through standard web technologies. A global trusted broker with a public IP address is designated as the mediation mechanism between the remote users and the home network. User access and permissions to particular services are configurable through the
middleware. The design supports services running on mobile devices in the home and therefore it preserves user-content privacy. This design addresses one aspect of pervasive connectivity, namely the NAT traversal, even though this solution requires that port forwarding is enabled on the local home router. Mobility and IP address changes are not addressed in this design.

A similar concept of a public mediator for the distributed execution of services is adopted in the SCOTT middleware [76]. The core of SCOTT is a set of modules that maintain service descriptions and service compositions and execute them across distributed agents. Service providers submit these descriptions to the SCOTT that categorizes them in a local knowledge-base. When clients request a particular composition, SCOTT retrieves the corresponding composition plans from the knowledge-base, creates an execution plan and submits them to the execution engine. The execution engine is a set of distributed agents residing on public Internet nodes, each executing part of the composition. Although this execution is carried out in a peer-to-peer fashion, the agents need to be publicly available. Therefore, the composition of public and private services across personal and mobile devices is not achievable.

The design in [77] describes a mechanism for utilizing the Cloud as a Service Discovery tool, similar to the functionality of UDDI. Smart-home network-enabled devices utilize gateways to advertise their capabilities with WSDL documents posted on the Cloud. Users from other homes can query the Cloud, acquire the WSDL documents and invoke services across homes. With the same model the Cloud application providers can offer applications and services to home users through the Cloud. For all this to be possible, this system modifies the Service Layer of a conceptual Cloud layered architecture and adds a Service Interface, Control and Directory layers for processing user requests. Although this design claims that the communications between homes are peer-to-peer, it does not provide a concrete solution to the pervasive connectivity
issue. Moreover, although services from the Cloud are offered to the home network, there is no provision for service composition using private and public services.

The Mobile Web Services Mediation Framework (MWSMF) system combines the Cloud and the P2P technologies to enable Web Services on mobile devices [78]. The MWSMF acts as an intermediary between the Mobile Hosts and the Web Services and it is configured both as a standalone entity as well as to run on the Cloud. Being on the Cloud enables it to adjust to the mobile operator’s network loads and offer better scalability. Discovery in MWSMF is realized through the JXTA P2P technology. Although this design does not target service composition, it does provide the potential to combine private and public services. It does not, however, target mobile devices or the Cloud.

The work in [79] proposes a peer-to-peer model connecting distributed OSGi containers [80] for composing services across devices. Components across these containers are interacting through an SOA augmented by a Mobile-Agent (MA) technology [81]. In particular, a set of agents facilitate the communications between various layers in the architecture. A Web Services Gateway bundle is also utilized to enable bundles from OSGi-based containers to interact with services from non-OSGi platforms. Discovery of services across containers is performed through an Agent Directory service, which is a customized home UDDI registry. Clients query the Directory service and then send an agent to represent them to a discovered service provider. The agent becomes local to the remote service provider and all interactions are therefore local. This design does not specify the P2P layer and how it deals with pervasive connectivity. In addition it is entirely based on OSGi bundles, which restricts its usage in other runtime environments.

The goal of the design described in [82] is to define a middleware architecture that enables a lightweight variation of the traditional Web Service, called the Mobile Web Service (MWS) that
can exist outside the traditional container. MWS-type service utilizes a library called Portable Middleware library (PM). MWS can either be incorporated into existing applications or executed as stand-alone applications. The middleware framework, labeled Web to Peer (W2P), deals with addressing through a naming service agent and manages the messaging between peer services. Finally it offers an event-based mechanism that is subscription-based – if a peer wants to receive a message from a certain source it subscribes to for those particular events. Overall this SOA achieves a platform-independent integration of services based on a variant Web Service technology. The design does not offer public and private service composition and it does not utilize the Cloud for discovering services.

There are also a number of additional solutions that introduce the P2P paradigm to the service level through a middleware architecture. The Wings middleware [44] deals with the provision of services from mobile devices operating in heterogeneous networks. It achieves this by utilizing a flexible middleware architecture that can be updated dynamically at runtime with plug-in adapters. These adapters allow mobile devices to operate over different overlay networks, such as JXTA, or peer-to-peer device connectivity architectures, such as UPnP, in order to discover services. Two more designs in this category are the Self Serv environment [83] that presents a layered middleware architecture for provisioning Web Services over a P2P network without a centralized controller, and the p2pWeb project [84] that utilizes web standards and proprietary technologies to deploy services over structured P2P overlay networks. In addition, p2pWeb handles addressing, naming and mobility by using specialized Uniform Resource Locators (URL) [85] through a proprietary Naming Service. These designs do not utilize the P2P overlays for accessing and composing private services but instead they use them for either general
Table 1.1 (1): Comparison of existing service composition middleware designs with the p2pSOA middleware solution.

<table>
<thead>
<tr>
<th>Project</th>
<th>Private and Public Service Composition</th>
<th>Mobile services</th>
<th>Pervasive Connectivity Solutions</th>
<th>User content privacy</th>
<th>Service discovery through the Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2pSOA</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chakraborty et. al. [65]</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Dahlem et. al. [67]</td>
<td>✓</td>
<td>Mobile clients only ✓</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>p2pWeb</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Antila [72]</td>
<td>×</td>
<td>✓</td>
<td>Partial</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Belimpasakis et. al. [75]</td>
<td>×</td>
<td>✓</td>
<td>Partial</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Wings</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Self-Serv</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>NEMO</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>MicroMAIS</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>MoSCA</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>CoSP2P</td>
<td>×</td>
<td>✓</td>
<td>×</td>
<td>(Not clear)</td>
<td>×</td>
</tr>
<tr>
<td>Wei et. al. [77]</td>
<td>×</td>
<td>×</td>
<td>(Not clear)</td>
<td>(Not clear)</td>
<td>✓</td>
</tr>
<tr>
<td>SCOTT</td>
<td>×</td>
<td>× (Possible)</td>
<td>×</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>MWSMF</td>
<td>× (Possible)</td>
<td>✓</td>
<td>× (Possible)</td>
<td>(Not clear)</td>
<td>×</td>
</tr>
</tbody>
</table>

Composite service orchestration for enhanced execution scalability or for gaining simple discovery and availability of services through Web browsers. Also they do not augment the Cloud capabilities in their design.

Finally, we describe a design that although it does not define a middleware layer as all the aforementioned projects, it achieves the integration of P2P overlays with SOA. The NEMO [86]
service orchestration framework achieves this integration utilizing location, service description, and peer trust verification abstractions over P2P for service provision. It incorporates simple and complex services originating from any type of device, any type of interface (legacy) and any type of discovery protocol. However, unlike our p2pSOA, NEMO does not present a middleware layer that separates service implementation concerns. Instead it utilizes customized Web Services standards for making services available for composition.

A comparison of the designs that are most relevant to p2pSOA is provided in Table 1.1. The criteria we used above to analyze these designs are shown across the top row. We grouped the designs as close as possible based on the axes they address. These designs provide novel solutions and features that partially address our requirements. None of the solutions covers all of these requirements in a single design. This result has motivated our work for designing a new middleware architecture to enable the composition of public and private services across heterogeneous devices regardless of their network location while handling mobility, all issues of pervasive connectivity and preserving user-content privacy.

2.5. The p2pSOA Unique Features

We outline here the unique features of the p2pSOA middleware architecture based on the comparison with the designs described above:

**Composition of private and public web services and the service programming model**

The p2pSOA middleware discovers services residing on distributed nodes based on textual composition descriptions. When all services in an intended composition are found, the p2pSOA middleware generates the necessary module for accessing them. This module contains the
invocation methods to the actual service implementation (e.g. Ajax/HTTP [87], SOAP/HTTP). The generated code does not include the business logic of the composite service. Separating the generated communications code from the composite service logic enables the developers to implement composite services based on abstract names which are not bound to particular service type or implementation. The developer does not need to know if the service is a SOAP-based or RESTful service or if it is private (behind NAT) or public. This is only determined at the end of the discovery phase: if a service is not found on a peer then it is assumed to be a public service. We describe further below this unique mechanism of involving the Cloud in the discovery phase in order to identify such services that fulfill the composition intent. In order to enable automatic generation of communication code that can execute also on mobile devices that lack compilation capabilities we generate scripting code (e.g. Ajax and Javascript [88]) that can directly execute on browsers and WebViews [89].

*Collaboration through Service Composition*

The p2pSOA design extends the concept of the trusted group offered by the peer-to-peer technologies to enable the creation ad-hoc communities. Members of the community volunteer their services to the other members in order to promote collaboration through service composition. This provision of services is possible both on personal computers and mobile devices by deploying Web Servers at each edge node of the trusted group and interfacing the nodes using standard Web Services protocols. The middleware accesses local databases residing on each edge node in order to retrieve the WSDL files of the services executing on that node. This feature eliminates the need for global and centralized registries for service discovery.
Cloud and P2P integration

The p2pSOA brings the benefits of Cloud capabilities into a peer-to-peer network. This subsequently extends the traditional Web Services model to enable the Service Providers to offer value-added services to the community of peers through familiar Cloud applications (e.g. Dropbox [11]. This unique feature gives to the Service Providers a role to play in ad-hoc formed P2P communities.. This model also extends the existing discovery methods by seamlessly integrating Cloud with P2P overlay services. In p2pSOA, when a discovery session identifies that a service is not available at a particular edge peer, the middleware accesses the Cloud shared directory of the particular peer. This enables peers to suggest alternative solutions to services. In that shared directory the peer may include links to alternative web services that reside in the public domain that can substitute for the missing functionality.

Service-level naming and addressing

The p2pSOA was designed to handle naming and addressing of devices residing behind NAT configurations. This is accomplished through a unique method that intercepts WSDL documents when they are transported during discovery from the server to the client. Once they are intercepted the middleware modifies them to include the server information (unique Peer ID) that is used on the client side for routing the service calls to the appropriate service implementation during the composite service execution. This mechanism eliminates the need for intermediary nodes to carry out the routing.
**Seamless integration of 3rd party technologies**

A number of existing technologies are adopted in the p2pSOA layered architecture. The 3rd party technologies are interfaced using Adapters that are particular to that technology. For example we implemented a P2PA Adapter that interfaces with the JXTA peer-to-peer technology. This enables us to utilize different technologies at the transport layer without modifying the technology or the p2pSOA middleware.
Chapter 3

The p2pSOA Architecture Design

In order to be able to combine private and public services into versatile composite services we need to address the problems of: (i) pervasive service connectivity, (ii) exposing services of personal devices across distributed peers, (iii) sharing these services within a group of trusted peers, (iv) executing them in a distributed fashion. In this chapter we outline the design of the p2pSOA middleware concept that provides the solutions to these problems. We first present a set of motivating applications that formulate the requirements for the new middleware. Subsequently, we analyze these design requirements and build the software model based on them. We then define the architecture layers and identify the main components of the p2pSOA architecture. Finally we present the functional aspect of the architecture and how its components collaborate to fulfill the vision.

3.1. Motivating Applications

The functional aspects of the first application are depicted in Figure 3.1. This application envisions a scenario where phones located in different home networks can collaborate to share pictures taken at real time. Alex visits his cousin in another town and wants to show him pictures of his new baby. Alex sends an SMS message to his wife to position the phone over the baby’s
crib and once she’s ready he initiates the service call from his phone. The remote phone enters a preview mode for a few seconds giving Alex’s wife the chance to adjust it to point at the baby and then the phone begins to snap pictures and sends them to Alex’s phone for his cousin to view in real-time. After a number of pictures are viewed, Alex stops the preview on the remote phone and ends the session.

The Unified Modeling Language (UML [90]) sequence diagram in Figure 3.2 describes the sequence of events during the execution of the application. When Alex wants to launch this application he brings up the p2pSOA WebView application. This application is a Webkit-based [91] browser for the Android operating system [92]. The WebView displays an HTML/Javascript [93] form that enables the user to execute the application. After he presses the Execute button in Step 1, the Javascript Client sends a service invocation to the snapPic Web Service executing on the remote phone that is acting as the server. The snapPic service calls a local Android Activity application [94] in Step 2 that subsequently puts the phone’s camera into a preview mode in Step 3. After a preview delay of three seconds in Step 4, the camera snaps a
picture and sends it back to the calling script (Steps 5, 6, and 7). If Alex presses the END button before the picture arrives (Step 9a), a request is sent to shutdown the preview mode on the remote phone. Once the preview mode is exited, the user is notified of the action in Step 12. Otherwise, Step 9b is executed and the process continues and a new picture is snapped.

The second application, described in Figure 3.3, envisions family members getting organized in preparation of a family dinner. Sofia at home wants to identify which of the family members is closest to a grocery store to pick up a few items needed for the dinner. She uses her laptop and launches the Family Errands application. The application contacts each of the family members’ GPS-enabled cell phones, retrieves their current location, and determines which cell phone is the one closest to which store. It subsequently sends a notification to that phone. When
the family member who owns the phone accepts the notification, a Google Map depicting the current phone’s location and the closest store is displayed along with the grocery list on that phone.

The UML sequence diagram in Figure 3.4 describes in more detail the execution of the application. Similarly to the previous application, when Sofia wants to launch the application she launches the p2pSOA WebView application. After she presses the Execute button in Step 1, the Javascript Client sends service invocations to the famErrands Web Service running on each of the family members’ phones. In steps 2 through 6, the Web Service calls a local Android Activity that access the on-device GPS and retrieves the current location of the device. Once all of the coordinates are retrieved, the application sends the coordinates to the Google Maps API service [95] in Step 7 and receives back a map of the locations of all the family members. Next,
in Step 8, the application sends the coordinates and a request for the closest stores within a user-defined radius (e.g., 1 mile) to Google Maps Places API service [96]. It then uses the returned distances matrix to determine which phone is closest to a store in Step 9, and it sends it the grocery list and the coordinates of the closest store in Step 10. In Steps 12 and 13, the famErrandsNotif Web Service on the phone contacts Google Maps API service with its coordinates and it subsequently receives and displays a map of its coordinates and the coordinates of the closest store. In addition, the shopping list is displayed. Finally, in Step 14, the remote phone sends a confirmation notification to the client phone that the list has been accepted.

The third application is described in Figure 3.5. This application is described as an extension of the Family Errands application. Although the functionality of this application is transparent to

Figure 3.4 (8): The UML sequence diagram for the Family Errands application.
the user, it is vital to the user experience. As the figure shows, the application incorporates the Cloud into the functionality of the scenario. When Sofia launches the famErrands application, a mechanism needs to be in place to handle cases where any of the family members’ phones is not executing a particular service. In such a case, the phone responds back with its Cloud ID that it can be used to access the peer’s online directory (i.e. Dropbox). The application can then be redirected to a particular server that hosts that web service and seamlessly invokes that service. If the service is not found there, the middleware searches the default Service Provider’s online directory for the same service.
3.2. Requirements for a new middleware architecture

Based on these motivating applications and on the requirements they set forth, we define the Use Case model diagram for the new middleware in Figure 3.6. In the first Use Case the primary actor utilizes the system to discover services. Services not found on peers are found on the Cloud. Another Use Case for the system is to execute compositions of services. These compositions involve executing either private or public services. There are two additional Use Cases for the system: the capability to stop the execution of a composition and to notify a secondary actor for a particular action.

The Use Cases expose a set of challenges from which we derive the following design requirements. First, accessing the services of personal devices from outside their local network requires solving the problem of pervasive service connectivity, including mobility and IP address changes. Therefore, if both clients and servers are mobile devices, the client should be able to connect to the server regardless of his/her location. Another consequence of this requirement is that mobile servers should be able to offer services to their trusted groups and community.
Next, a mechanism is necessary to make these services available for discovery and composition with public services across distributed peers. The issue of trust is also of great importance. In both applications, the peers want to share their information (baby pictures, locations) only within their trusted group and therefore they should be able to form and join this group requiring membership authentication. This requirement necessitates the capability to create dynamic ad-hoc communities that will enable its members to seamlessly collaborate through service compositions.

The requirements gathered from the third application, define a new role for the Service Provider (SP). The SP may be interested to offer value-added community services through the Cloud. The members of the community can subscribe to these services that can be downloaded and executed on their devices. In addition, it extends the discovery capabilities of the middleware and enables community users to offer recommendations to the rest of the members.

In order to address the above requirements, we need to define a new middleware architecture that allows the sharing and execution among trusted peers of service compositions combining private and public services using both personal computers and mobile devices.

### 3.3. A Service-Oriented Architecture based on Web Services

A general four-layer SOA architecture that supports Web Services computing and can enable the above vision is shown in Figure 3.7. The advantage of using a layered architecture is that it encapsulates and groups entities that are cohesive to the concerns of a layer [27]. For example, the Transport Layer includes both Internet and Peer-to-Peer communication protocols. This offers a clear separation of concerns at each layer that ultimately yields modular designs.
The lowest layer of the architecture encapsulates the world of the Distributed Services layer. This layer encompasses the services provided by various hosts. The next layer up is the Transport Layer that groups the mechanisms necessary to connect the distributed services. If the service is public then the provider of the service is directly accessible through the Internet over standard protocols such as the HTTP protocol. However, if the provider is a peer in a trusted group residing behind NAT, then NAT traversal mechanisms, such as a P2P overlay, are necessary to provide access to the web service and address the pervasive service connectivity issues between the consumer and the provider.

The top layer in the architecture is the User Interaction (UI) layer. This layer represents the end-user programming tools domain that enables the user to build service compositions. The user

Figure 3.7 (11): A layered architecture supporting Web Services composition and execution.

<table>
<thead>
<tr>
<th>User Interaction Layer</th>
<th>End-User Programming Tools</th>
<th>Elementary Services Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime Layer</td>
<td>Runtime Environments (JRE, Android, OSGi, other)</td>
<td>Composite Service (CS)</td>
</tr>
<tr>
<td>Transport Layer</td>
<td>P2P Overlay Network (JXTA, UIA)</td>
<td>Internet</td>
</tr>
<tr>
<td>Distributed Services Layer</td>
<td>Services (iJetty, OSGi, WS other)</td>
<td>Remote Web Service A (private: iJetty, OSGi)</td>
</tr>
</tbody>
</table>
Figure 3.8 (12): The p2pSOA middleware layer incorporated in the general Web Services architecture.

can utilize workflow programming tools to describe a Composite Service (CS) business logic and the elementary services that compose it. The Runtime Layer encapsulates the service compositions that are generated in the User Interaction Layer and their execution environments.

3.4. The p2pSOA middleware architecture

The Web Services architecture shown above supports composition of services and their execution but it imposes the requirement that all services must be publicly addressable. In order
to enable access to private services there is a need for mechanisms that can discover and connect
private services with each other through the P2P technologies and with public services through
the Internet. In addition, this mechanism should be able to facilitate the execution of the resulting
compositions transparent to the client.

We designed p2pSOA, a middleware layer that fits in the above architecture, that facilitates
the sharing and distributed execution among trusted peers of composite services consisting of
private and public services using both mobile and static devices. The p2pSOA shown in Figure
3.8, mediates between the Runtime and the Transport layers of the above general architecture
and offers discovery and execution services.

The p2pSOA enhances the general Web Services model with the discovery of private and
public services and the execution of composite services. For this purpose, it defines adapters and
agents. Adapters offer a level of indirection with the entities they adapt to. They separate the
various application concerns, for example the peer-to-peer communications code, from the
service implementation. Specifically, the P2P Adapter allows us to incorporate different P2P
overlay technologies (e.g. JXTA, UIA [97]) without modifying the core of the middleware. For
instance since JXTA’s rendezvous routing scheme does not exhibit optimal scalability, it may be
necessary to replace it with another overlay for certain applications. Another example is the
Runtime Adapter that enables any service to be adapted to execute on a variety of CS runtimes
(e.g. OSGi, Python [98], Java Runtime Environment (JRE)). While the adapters enable local and
remote private services to be involved in composition, the agents provide the means to locate
these resources and to carry out their execution. For example, the Discovery Agent is responsible
for acting upon configurable parameters in order to choose the appropriate services to build the
CS. Similarly, the Execution Agent’s responsibility is to route the calls to these services through the overlay without requiring any modifications to the CS.

### 3.5. The Middleware subsystems

As mentioned above, the p2pSOA middleware layer is composed of a set of subsystems. The adapters which deal with the issues of pervasive service connectivity and distributed group management and also generate stubs that can execute in various CS runtime environments. It is also composed of two agents that use the adapters in order to perform the discovery and execution of the CS respectively. Figure 3.9 depicts the dependencies between these subsystems at the architecture level. As it can be seen, both the Execution and the Discovery Agent depend on the Peer-to-Peer Adapter for communications across peers. The Runtime Adapter depends on the Discovery Agent for information on generating the appropriate stub modules. We map these dependencies in the layered architecture when we describe the Discovery and Execution phases.
later in this chapter. In the next sections we describe these major entities of the middleware and their roles in more detail.

The Composite Service (CS) is the entity that initiates the execution in the p2pSOA design. The CS may combine both private and public elementary services into more complex tasks. It is developed by expert programmers and it is offered to the user through the Service Provider. The CS is developed using the abstract service names to denote calls to elementary services. The abstract service names are an indirection pointing to the communication stubs that are generated by p2pSOA.

The user initiates discovery and execution sessions through containers such as standard browsers, OSGi containers and mobile browsers based on WebView. The choice of such tools depends on the implementation of the CS. Generally browsers, either standard or mobile, execute scripting languages such as JavaScript. Composite services implemented in Java are either executed through the JRE or containers such as the one defined by the Open Gateway Services Initiative (OSGi) [80].

We selected to also support OSGi because it is a popular component model for enterprise and residential service oriented applications. The OSGi specifies a component model and a container for service-oriented applications and residential services. Application software offering services are deployed in OSGi in the form of software “bundles” that can be installed, started and terminated, configured, and updated on the fly (at runtime) without the need to reboot the container. These services can be discovered by other bundles in the local registry and combined to form composite complex services. Devices in the local network connect to the gateway and expose their services in the OSGi’s registry. Other devices can thus discover and utilize them giving rise to a pervasive-like SOA model.
3.5.1. The Peer-To-Peer Adapter (P2PA): Routes messages between endpoints in the P2P trusted group

The P2PA offers a layer of indirection allowing the use of different P2P overlays. At initialization, the server-side P2PA is responsible for generating and storing in permanent storage a unique peer ID called the Peer Unique Identification (PUID). The reason for storing the PUID is for cases where a CS may be executed at a different time than the time a service is discovered. During that period peers may reboot and their PUID would be otherwise re-generated causing stale discovery results to exist on client peers and therefore rendering their CS invalid.

At discovery time, the client-side P2PA forwards the discovery queries to the server-side P2PA through the P2P overlay. The server-side P2PA forwards the queries to the local WSDL repository and receives back the path in permanent storage where the appropriate WSDL file resides. Subsequently, the server-side P2PA loads and modifies the WSDL file before sending it back to the client-side. When the response returns, the client-side P2PA first retrieves from the P2P overlay the P2P endpoint from which it arrived. Then it retrieves the PUID of the server-side peer from the WSDL file in order to update a map table, called the Message Routing Table (MRT), that maps the PUIDs to endpoint identifiers understood by the P2P layer. Subsequently it forwards the PUID to the Runtime Adapter that generates the stubs for the CS.

During execution, the client-side P2PA intercepts the outgoing SOAP or REST message originating from the CS, retrieves the PUID from it to use it for looking up the corresponding P2P endpoint in the MRT, and forwards the messages to the server-side peer. The server-side P2PA is then responsible to forward these messages to the local Web Server. For the cases of SOAP messages, the P2PA is responsible for reversing these SOAP modifications before passing
Figure 3.10 (14): Discovery Phase for discovering a private Web Service.

the messages to the server. This is necessary to keep the Web Services that the p2pSOA interfaces intact without any modifications to their original implementation.

3.5.2. The Discovery Agent (DA): Enabling service discovery

The client-side DA is responsible for reading the atomic service descriptions submitted by the User Interaction layer, generating discovery queries for them and forwarding these queries to the client-side P2PA. As shown in Figure 3.10, The client-side P2PA sends these queries to each of
Figure 3.11 (15): Discovery Phase for discovering a public Web Service.

the known peers in the trusted group. The server-side DA is responsible for resolving the client-side DA queries to WSDL files in the local WSDL repository, and responding if there is a match. Once all of the WSDL files are returned, the client-P2PA forwards them to the client-DA that in turn sends them to the Runtime Agent (RA) responsible for generating the appropriate client stubs. We discuss the RA later in this chapter.
3.5.3. Service Discovery and Cloud Integration

If a service necessary for the CS to execute is not found on a peer, the DA looks to the Web to find it as shown in Figure 3.11. The peer not running the particular service, returns a Cloud URL to the client-side DA. This URL is a public address that identifies a directory in the Cloud (e.g. Dropbox.com) where the peer maintains a mapping between abstract service names and URL links to public services with functionality identical to the service that is missing. The Cloud URL includes a Cloud ID which is a unique ID assigned to the peer by a Cloud administrator and which the peer is willing to share within the trusted group.

If the service is not found in the peer’s Cloud directory, the DA uses the Provider’s Cloud URL to access the directory in the Cloud where the Service Provider maintains such a mapping. The Provider’s Cloud URL is available to the peers when they acquire atomic or composite services from the Provider. If the services is not found there either, the DA aborts the Discovery and notifies the User.

This mechanism enables peers to recommend alternative solutions to services that are not currently executing on a particular device. With a similar token, it gives the Service Provider the capability to offer valued-add services to the community. Finally, it preserves the same p2pSOA model for discovering both public or private services.

3.5.4. The Runtime Adapter (RA): Automatic generation of stubs

Once the DA retrieves the WSDL documents from the P2P layer in response to its queries, it forwards them to the Runtime Adapter (RA) to generate the client stubs. The RA offers a layer of indirection allowing the use of different CS runtimes. The RA generates on the fly the service invocation client stubs for a particular CS runtime using the corresponding WSDL files. The
client stubs are either Java classes, in the case of SOAP services, or Web-based scripts that include the language bindings and API calls necessary for the client application to call a remote Web Service. On personal computers, the RA compiles on the fly the generated Java classes for SOAP invocations. Calls to the remote service methods are first routed through the client stubs that in turn generate the appropriate messages that our middleware routes to the remote service implementation. The client stubs process the responses from these calls and return the results to the calling application. Only the WSDL files for the services that are involved in the CS are used to generate the corresponding stubs.

By adopting the SOA/WS model of execution that separates the communications code (client stubs) from the application logic, p2pSOA allows the CS to be platform independent and offers ease of creation to the developer. The CS can be executed on a different choice of runtime based on a set of criteria, such as speed or memory requirements. By generating stubs and separating them from the CS implementation, the developer can produce the CS without explicitly using the names of the services in the code.

3.5.5. The Execution Agent (EA): Composite service deployment

The Execution Phase event flow is shown in Figure 3.12. During the CS execution, the CS calls the client stubs that generate SOAP and REST calls to the remote services. As mentioned earlier, in the p2pSOA model the actual calls to any type of service, private or public, are handled by the stubs and are transparent to the CS.

For private services, the EA intercepts these calls and forwards them to the client-side P2PA. The P2PA uses the p2pSOA routing mechanism to transport them to the appropriate server-side peers. The server-side P2PAs forward the messages up to the server-side EAs, which
in turn forward them to the local web services, in this case Web Service A. The responses of the service invocations are returned back to the client-EA and from there to the client stubs and the CS. Depending on the CS implementation, the user is notified of the successful execution of the CS.

For services that reside on the Cloud or on public Web Servers, for example Web Service B in Figure 3.12, the requests are not mediated through the p2pSOA middleware. The stubs contain calls to public IP addresses and therefore these invocations are performed using standard
Web technologies protocols (e.g. HTTP). The responses of these service invocations are handled with the exact same manner as the ones from the private services.
Chapter 4

The p2pSOA Implementation

In this chapter we outline the full implementation of two instances of the p2pSOA software architecture defined in Chapter 3: the standard p2pSOA that targets rich devices such as laptops and desktops; the Mobile-p2pSOA that incorporates mobile devices such as cell phones and tablets into the composition and execution of services. We first outline the design and functionality of the User Interface to the p2pSOA middleware that is common to both instances. Next we describe the unique features for each of the two instances as well as the commonalities in their implementation. The rest of the chapter is devoted to the implementation of the Discovery and Execution Phases. The implementation includes the integration of the Cloud in the discovery phase, and the execution of private and public services in the same composition in the execution phase.

Both instances were implemented in two different flavors of the Java programming language [99]. The standard p2pSOA was implemented in Java Standard Edition (JSE) whereas the Mobile-p2pSOA was implemented in Android Java. The Android Java [92] includes core libraries that offer most of the functionality in the JSE.

Java is a popular and familiar language with syntax similar to the C/C++ programming languages [100]. It is a general-purpose object-oriented language that has a wide range of usage
including in the area of web technologies. It is designed to hide many of the low-level details from the user, such as memory management and garbage collection. Java is compiled into an intermediate code called bytecode, which allows applications written in it to be platform independent and be deployed on any device that can execute a Java Virtual Machine (JVM). This property enables p2pSOA to execute on any java-enabled platform, personal computer or mobile.

### 4.1. The User Web Interface (UWI)

The p2pSOA middleware utilizes a front-end user screen implemented in HTML and a set of background JavaScript modules, which are common to both implemented p2pSOA instances. However, the containers that can be used to load and deploy the front end differ between the two instances. The p2pSOA on personal computers is displayed through standard browsers, whereas in mobile devices through the WebView UI. We describe this in more detail in the sections to follow. As shown in the leftmost pane of Figure 4.1, the Discovery screen includes a DISCOVERY button and a input text box. The button initiates a discovery session whereas the input text box is used to enter the name of the Composite Service (CS) that is valid for this session. The CS resides in the p2pSOA distribution directory. When the discovery phase is completed, the p2pSOA middleware notifies the user with an alert that the generation of the client stubs was carried out successfully as shown in the middle screen. The User confirms by
pressing the alert window. This action generates the Execution screen shown in the right page of Figure 4.1. This page displays the EXECUTION button that initiates the execution session. Depending on the CS implementation, any subsequent alerts to the user or any results are displayed in this screen.

Figure 4.2 shows the interactions between the various modules involved in the Web Interface System. The windows in gray denote what the p2pSOA offers or generates and in white the entities that the user obtains from the Service Provider. First the User launches the p2pSOA.html HTML page that draws the Discovery screen. When the Discovery button is pressed, the p2pSOA.html page calls the Discovery() function in the Discovery javascript module (Discovery.js). This function reads the findServices.txt file that contains a list of the abstract names of the services involved in the particular composite service and it generates a
function Execute_Composite() {
    alert("Executing the Family Errands Composite Service");
    numOfPeers = famErrands();
    allCoords = createArray(numOfPeers,2);
    allDistances = createArray(numOfPeers,2);
    getCoords();
    getPlaces();
}

function famErrands() {
    famErrands_private("2FBD56B65BD84494BDD1E1C0915C6F92");
    famErrands_private("D266F33D554A4A55B5131243E3CCC243");
    famErrands_public("E1B2A02D36EF46A180E9CCFED633EAC2");
    return(numOfPeers);
}

Figure 4.3 (19): The Family Errands Composite Service and Stub module.

query call that the p2pSOA DA intercepts and sends to the peers in the trusted group. Once all
discovery results are received, the p2pSOA middleware notifies the Discovery.js that in turn
notifies the User about the event. Once the User confirms by pressing the alert window, the
module generates the AtomServ.js stub module and it draws the Execution button on the main
HTML page. Subsequently when the Execute Composite button is pressed, the function
Execute_Composite() in the JavaScript CS.js module is called. The CS.js (shown in
Figure 4.3 in more detail and discussed in the next section) calls the services with their abstract
names that in turn make calls to the JavaScript AtomServ.js module. This module is the stub
module that contains the actual calls to the remote services. Once the results of the service invocations return, the CS.js draws them on the main HTML page.

4.2. The Composite Service and the Stubs

The Composite Service (CS) is generally implemented by experienced programmers. It can either be a Java executable (e.g., OSGi bundle) or a JavaScript module. A segment from the CS used in the implementation of the Family Errands application is shown in Figure 4.3. The CS developer is aware of the service naming convention employed by the p2pSOA. He/she developers the CS with calls to the services through these names without the need to be aware of the actual service name at the server node. In the example shown, the module in the top pane is the CS. As mentioned in the previous section, the function `Execute_Composite()` is the main function of the CS. This function is called by the Discovery.js module once the User presses the Execute Composite button.

The CS includes calls to the services through their abstract names. As shown in Figure 4.3, the call to the famErrands service is actually a call to the `famErrands()` function in the AtomServ.js stub module that is generated by the Runtime Adapter (RA) after the discovery phase is completed. The AtomServ could include calls to both SOAP and REST-style services. The p2pSOA middleware inspects the various WSDLs from the peers and according to the type of the WSDL it generates different functions in the AtomServ to handle these two types of services. For the REST-style services it generates standard Ajax calls whereas for the SOAP services it utilizes the Prototype.js [101] and Ws.js [102] JavaScript libraries. In the example in Figure 4.3, it includes two Ajax calls to a private REST service on two peers in the trusted group.
It also includes a call to a public REST service that was discovered through the Cloud, since it was not available on the particular device.

The AtomServ.js module includes both synchronous and asynchronous calls to the remote services. Both such calls are generated for the same service by the RA. In addition, the RA generates both multicast and unicast call (to a single peer) for each service entry in the CS. The JavaScript-based AtomServ module is compatible with both the standard p2pSOA and the Mobile-p2pSOA implementations. However, the standard p2pSOA implementation can also handle Java-based clients. In such cases the RA calls the WSDL2Java tool [103] to generate the appropriate Web Service stubs and then compiles them on the fly.

4.3. The standard p2pSOA

The standard p2pSOA supports the execution of composite services that are implemented in JavaScript. JavaScript [88] is a prototype-oriented object-oriented scripting language. It is considered the primary scripting language of choice for the Web and it is supported by most standard web browsers. JavaScript adds dynamic behavior and thus enhanced user interfaces to HTML-based web sites. We chose to support scripting in order to enable automatic stub creation on mobile devices that do not have compilers.

The server-side component of the standard p2pSOA can interface full-featured SOAP-based web servers, such as Apache Axis, or web servers that support the Java Servlet technology [104] and can host REST-style services. The Apache Axis project implements a web services container for SOAP services. The standard p2pSOA was tested with services supporting SOAP both as OSGi bundles and as plain Java classes on the Apache Tomcat Web Server [105]. The OSGi bundles are installed and deployed in OSGi containers either through the OSGi Desktop UI or
through the Console Command Shell. The standard version of the middleware was tested with the Knopflerfish [106] OSGi 2.3.4 implementation.

The Servlet technology [104] extends the Web Server functionality with dynamic behavior. The incoming requests are handled by the Servlet through a separate thread. Although Apache Axis2 [103] currently supports REST-style Web services as well, the standard p2pSOA was tested with a set of REST services implemented as Servlets on the Eclipse Foundation’s Jetty Web Server [107].

The WSDL files for the Web Services are stored in databases on the server. On personal computers we implemented a WSDL database on the mySQL database engine [108] and on Android on the SQLite database engine [109]. The database table schema we defined declares two columns: the abstract name of the service and the path to the WSDL file in the local storage. The Discovery Agent (DA) connects to the database upon startup and issues SQL queries using the abstract names of the service with every incoming discovery request. Therefore a naming convention is imposed by the Application Vendors to match the abstract names in the CS with the service names stored in the databases on each peer.

4.4. The Mobile-p2pSOA

The Mobile-p2pSOA implementation supports Android-based mobile devices. The Google Android Operating System [110] is a versatile software stack and environment for deploying complex mobile applications. It is built on top of the Linux 2.6 kernel and it utilizes the Dalvik Virtual Machine as part of its runtime. Android has seen a tremendous increase in popularity recently and it is the major competitor to the Apple/iPhone iOS [111]. One of the major factors for its popularity is that it is open source and it utilizes Java. Moreover, any commercial Android
phone can be used out-of-the-box for the development of applications without the need to “jailbreak” or “root” the device. These are popular techniques for providing root access to the device’s operating system with the ultimate purpose of gaining complete software control of the device and its hardware.

In the Mobile-p2pSOA, the discovery and execution phases are executed through the Android application p2pSOAWebV that launches the Android WebView class. This class is based on the open-source Webkit browser web engine. The Android WebView does not provide JavaScript support or widgets by default and therefore the p2pSOAWebV enables these properties at runtime. The UI in the mobile version of p2pSOA is identical to the screen used in the standard version shown in Figure 4.1.

Currently, the only full-featured Web Server with servlet support for the Android OS is the iJetty Web Server [112], a porting of the Jetty Web Server. It offers a user-friendly UI that allows the user to stop and start the server and to install and deploy web services. The Mobile-p2pSOA was tested with a set of REST-style Web Services implemented through servlets and deployed in the iJetty Web Server.

### 4.5. Common implementation

In both implementations, the P2PA entity was implemented to interface Oracle’s (previously a product of Sun Microsystems) JXTA peer-to-peer software stack as our supported transport mechanism. The JXTA technology [19] is an open-source overlay mechanism based on a set of protocols that hide the topology and nature of the physical network from the application developer. It defines six protocols that deal with peer discovery, group peer membership, connectivity binding, routing, and peer information. Peers can form trusted groups through
authentication in order to share common services and interests or to provide combined services to other peers or groups. Peers can discover other peers and their services using a special directory-type peer called Rendezvous, responsible for keeping track of other peers and groups in the network. The Relay is also another special type of JXTA peer that is responsible for forwarding messages between edge peers residing behind firewalls and NATs. Connectivity is offered by pipes, a similar concept to the UNIX pipes, which are based on TCP sockets and provide a mechanism for exchanging messages. There are two types of pipes based on the service they offer: server and client pipes. Similar to other resources in the JXTA domain, pipes are also assigned unique IDs. In the p2pSOA context, we name the unique ID of a server pipe *Server Pipe ID* (SPID). The SPIDs are used in the MRT on the client peers to map peers to server pipes.

JXTA is currently an established and well supported open-source peer-to-peer mechanism. It offers the capability to traverse firewalls and deal with NATs through its relay peer and also the ability to handle mobility (IP address changes) by utilizing the peer IDs in advertisements rather than the actual IP addresses of a peer. JXTA is an important technology extensively used by the distributed computing community.

Similarly to the P2PA, the RA is also common in both implementations for the REST-based services. It generates JavaScript and Ajax stubs that can be executed both by a standard browser as well as by the WebView UI class defined above.

### 4.6. The p2pSOA End-to-End Message Routing

One of the main goals of p2pSOA is to enable true peer-to-peer execution of composite services by combining private and public services over P2P overlay networks. The service invocations
are accomplished through REST or SOAP messages. Both types of messages use IP addresses and Domain Naming Service [113] (DNS) servers for service addressing as part of the XML documents they communicate. Since hosts may actually have private addresses that are meaningless in a global scale, one of the technical innovations of our work is that we are intercepting and manipulating on the fly these REST and SOAP messages in order to achieve end-to-end routing of service invocation messages, without changing the implementation of services or requiring service-level intermediaries and centralized service registries.

In our solution, we introduced a mechanism based on the PUID to help us deal with addressing. As mentioned in a previous chapter, our middleware maintains a routing hash table, the MRT, that maps the PUIDs to endpoint identifiers understood by the P2P layer. Examples of P2P endpoint identifiers are the SPID for JXTA and Endpoint IDentifiers (EID) for UIA. During discovery, the p2pSOA middleware on the server-side appends the PUID in the SOAP action names of the WSDL file of each elementary SOAP service, which is part of the CS, before the file is delivered to the client-side. Similarly, the PUID is appended in the Operation Name field of the WSDL file of each elementary REST service. The RA utilizes the PUID to generate the stub module. This results in SOAP and REST stubs at the client-side that include the PUID in the service invocation logic.

In Figure 4.4(a) we show a fragment of the WSDL description of the SOAP-style Books service, which is part of the BooksSearch application implementation (described in more detail in Chapter 5), before it is modified by the middleware. In Figure 4.4(b) we show the same fragment after it is edited. The service name and methods have a PUID appended to the end of the name. During the execution phase, the p2pSOA middleware should be able to intercept the SOAP messages originating from the client-side stubs without modifying the runtime. For this
purpose, the server-side also edits the IP address or DNS name of the namespace URL of the service in the WSDL file (Figure 4.4(b)). The address (e.g. 192.168.1.2) produced by a network implementing NAT, is changed to become the loopback address (127.0.0.1) and the TCP port of the namespace URL (8080) is changed to a port (25718) that our middleware reserves for this purpose. With these changes, the service invocations in the client-side stubs during the CS execution, generate SOAP messages that are actually sent to the localhost address on the port 25718. The p2pSOA client-side middleware is listening to this port on the localhost and intercepts these SOAP messages. It then retrieves the PUID from them and uses it as a key to look up in the MRT the appropriate P2P endpoint to route the SOAP messages. As shown in Figure 4.4, the WSDL file name is also edited on the client-side to include the PUID before it is stored in permanent storage. The reason for this is to avoid naming conflicts with WSDL files.

**Figure 4.4 (20):** WSDL fragments for the Books SOAP service: (a) the original WSDL file and (b) the modifications performed by the p2pSOA end-to-end protocol.
belonging to services that may have the same name but implement a different functionality. Finally, in order to preserve the mapping between the PUID and the P2P endpoints in host reboots or crashes, the MRT is also stored in permanent storage. For public services, for which SOAP routing is not necessary, the middleware does not edit their corresponding WSDL files. Therefore their generated stubs maintain their public IP addresses or DNS names.

A segment of the SOAP binding stub for a JavaScript-based client is shown in Figure 4.5(a). This stub is generated directly by the p2pSOA middleware and it is using the Prototype

---

**Figure 4.5 (21):** SOAP (top) and REST stub segments showing the runtime routing information.
Javascript Framework (prototype.js) and the Web Services JavaScript Library (ws.js). Java-based clients are generated with calls to the Apache WSDL2Java tool that is distributed with the Axis1.x/Axis2 Web Services engine development package [114]. As it can be seen, the addressing information that was entered in the WSDL document for the SOAP service in Figure 4.4 was included in the generated stub. During execution, the calls to the Books web service will be sent to the loopback address (127.0.0.1) and the TCP port of the namespace URL (25718). Figure 4.5(b) shows a segment of the generated Ajax stub for a JavaScript-based REST client. As it can be seen, in this case, the service invocation to a famErrands web service (also presented in more detail in Chapter 5) in the open() method of the XMLHttpRequest object is sent to the localhost and the TCP port 25718. The PUID is sent as a special HTTP header [59] with this call. HTTP headers are an integral part of the HTTP protocol and they are included in both requests and responses between parties. A special HTTP header is the X-header. The string included in the header is prefixed with the letter “X” in order to specify that this is a non-standard application-specific header. The HTTP protocol will ignore this header and simply pass it to the target application. When this call in activated during the execution phase, the client-side p2pSOA extracts the PUID from the header, looks up the corresponding SPID in the MRT and it routes the REST call to the peer.
4.7. The p2pSOA Runtime

At runtime the p2pSOA middleware enters a set of states for initializing the various entities, discovering atomic services, and executing composite services. We describe here the implementation of these phases in detail.

4.7.1. Initialization Phase

Before the p2pSOA middleware is launched on a personal computer or mobile device, the WSDL files of the Web Services hosted on the device are stored in the WSDL database. This is shown as Step 1 in Figure 4.6. In the current prototype, the WSDL files are entered manually. In our future enhancements we are planning on creating a tool that automatically enters the
documents in the database each time a Web Service is deployed in the local Web Server. The Web server on each device can be launched either manually or it can be configured to run as a background service when the device is powered up.

The Rendezvous/Relay (Rdv/Rly) peer is configured on an available public IP address and launched by its owner who is also a member of the trusted group. This IP address is a configurable parameter and it can be distributed to trusted peers through an out-of-band method (e.g. email, phone call, …). Upon startup, the Rdv/Rly creates a Trusted Group (Step 2 in Figure 4.6) and assigns a set of pre-defined authentication credentials to it. Subsequently it waits for connections and requests from Edge peers.

Once the User launches the p2pSOA middleware, the DA, EA, RA and P2PA are launched and initialized. The DA and EA connect to the P2PA through TCP sockets. The P2PA first generates the PUID for its peer and then contacts the Rendezvous/Relay using its aforementioned public IP address. The IP address is made available to the P2PA by the front end (launching scripts on personal computers or launching applications on mobile devices) of the p2pSOA middleware during the launching process. In Step 3, peers send the Rdv/Rly a request to join the group using the group’s name and the authentication credentials. Once granted permission, with Step 4 each of the peers generates a JXTA bi-directional server pipe (for their services) and they advertise it with the Rdv/Rly. During the Discovery Phase (presented in the next section), peers query the Rdv/Rly for the server pipe information of other peers. The Rdv/Rly returns the server pipe advertisements that the client peers can use to send connect messages to these server pipes. In addition, these advertisements include the SPID of the server pipes that are necessary for routing messages to the appropriate peers.
Peers in the p2pSOA design act both as servers and as clients. When acting as clients, the peers discover and connect to server pipes through JXTA Bi-directional client pipes (Step 5). The pipes are used to access services that are executing on a particular peer. In cases where the server-side peer moves (changes its IP address), the client-side peer can still locate the server pipe since it is advertised using its JXTA peer ID and not its physical IP address.
4.7.2. Discovery Phase implementation

The entities involved in the Discovery Phase and the interactions between them are depicted in Figure 4.7 and in the UML sequence diagram in Figure 4.8 respectively. Since the steps in Figure 4.8 show finer detail, their numbers do not necessarily match the step numbers in Figure 4.7. For clarity, the Rendezvous/Relay (Host 3) is not shown in the UML diagram. The figure shows a discovery session snapshot of a connection between two hosts, HOST1 and HOST2. HOST1 assumes the role of the client that initiates the discovery session while HOST 2 is the server. As mentioned earlier, peers in p2pSOA can be both clients and servers. The
Rendezvous/Relay peer runs on HOST 3. Both peers reside in separate local area networks with routers that are configured with a firewall and implement Network Address Translation (NAT).

In Step 1, the User of HOST1 launches a Browser (e.g. Firefox) or a WebView and loads the p2pSOA HTML client form. The Discovery Phase is initiated when the User presses the DISCOVER button in the form. The DISCOVER button initiates a Javascript/Ajax call to the local host on a user-defined port. In our implementation we used port 25722. The client-side Discovery Agent intercepts the call, loads the abstract service names from the findservices.txt file, and in Step 3 passes the service names to the P2PA. Subsequently, in Step 4, P2PA sends service queries using each abstract service name to all known server pipes that it is connected to, as listed in the MRT. Steps 5, 6 and 7 show the flow of messages between the two peers through the JXTA layers. The service queries are passed on to the server-side P2PA and then in Step 8 to the server-side DA. The server-side DA creates a database connection to the local WSDL database and it uses the abstract service names to query the DB.

If the WSDL file exists, the query returns a pointer to where the document is stored locally which the DA uses to retrieve the document. The response message then follows the reverse direction from the server to the client. Specifically, the DA returns the message to the server-side P2PA. Following the p2pSOA End-to-End Message Routing mechanism described in Section 4.6, the P2PA modifies the WSDL document to include the peer’s PUID and then sends it to the JXTA layers. The response message is transmitted back to the client-side P2PA. The P2PA there retrieves the PUID from the WSDL file and adds it to the MRT along with the SPID from which it originated, in this case HOST2. Subsequently it forwards the WSDL file to the DA that sends it to the RA. The RA inspects the WSDL document to determine whether the corresponding
Figure 4.9 (25): Activity diagram for the incorporation of Cloud services in the Discovery Phase.

A service is a SOAP or a REST service and it generates the appropriate stubs. In the former case, the RA executes the Axis WSDL2Java tool to generate the client stubs for Java-based clients, otherwise it creates JavaScript stubs for both types of services, SOAP and REST. The stubs are generated with service invocations to the localhost IP address (127.0.0.1) and a pre-defined port (25718). This is the port that the EA is listening to for outgoing service calls. Once all of the WSDL files are retrieved, either from peers or the Cloud (described in the next section), the p2pSOA HTML client form is updated and draws an EXECUTION button.
4.7.3. Cloud Discovery Implementation

The Discovery Phase utilizes Cloud services to find services that are not present on peers. We use the activity diagram in Figure 4.9 to describe this operation. If the discovery request described in the previous section does not return any results, the DA generates the NOTFOUND response. It adds the PUID, the Cloud ID URL and the string NOTFOUND and then forwards it to the P2PA. The message is then routed back to the client-side P2PA. The P2PA retrieves the PUID of the server peer and it adds it along with the NOTFOUND flag in the MRT. The DA scans the MRT and when it finds a NOTFOUND flag for an entry, it extracts the Cloud ID URL and the abstract service name from the message and it generates an HTTP call to the Dropbox.com web site. The call retrieves a default myServices document, that stores a list of service abstract names with URL links to their WSDL documents, and the DA scans it using the abstract service name of a service that is missing from a peer in the trusted group. If the service is found, the corresponding URL is used to retrieve the WSDL file. If not then the DA generates an HTTP call to the Service Provider’s default Cloud ID URL that points to the default allServices document. Similarly to the myServices document, this document stores links to services that the SP provides. If the WSDL is not found there either, the DA aborts the generation of the stubs and it notifies the user of the action. If the WSDL file is found in either location, it is returned to the DA that stores it in the MRT and instructs the RA to generate the stubs once all the WSDL documents are found.

4.7.4. Execution Phase implementation

The entities involved in the Execution Phase and the interactions between them are depicted in Figure 4.10 and in the UML sequence diagram of Figure 4.11. The steps in Figure 4.11 are
Figure 4.10 (26): The Execution Phase implementation

annotated with the corresponding event description and they do not match the step numbers in Figure 4.10. Also the Rendezvous/Relay (Host 3) is not shown for clarity in Figure 4.11.

In Step 1, the User initiates the Execution Phase by pressing the EXECUTION button on the p2pSOA HTML client form. In Step 2, the EXECUTION button executes the corresponding Execute_Composite() function in the CS that in Step 3 makes calls to the generated stubs for each service in the composition. In Step 4, the stubs send a SOAP/HTTP or REST/HTTP call destined for the remote service to 127.0.0.1:25718. The client-side EA intercepts the outgoing request message and passes it to the client-side P2PA in Step 5. Because the PUID was inserted in the WSDL files by the server-side P2PA during the discovery phase, and therefore the stubs
Figure 4.11 (27): UML sequence diagram for the Execution Phase.

included it in the outgoing messages, the client-side P2PA retrieves the PUID from the message and it hashes the corresponding SPID from the MRT table in Step 6. The P2PA uses the SPID to pass the message to the corresponding server pipe. In Steps 8, 9, and 10, the JXTA layers transports the message across the P2P overlay to the server-side P2PA. If the service is SOAP-based, and since in our design we used the Axis Engine unmodified, the server-side P2PA strips the PUID that was used for routing from the service method name in the SOAP envelope and passes the message to the server-side EA in Step 11. The EA then wraps it in a SOAP/HTTP call and then submits it to the Axis server in Step 12 as a standard SOAP message. If the service is a REST-style service, the incoming message does not include any modifications and it is simply forwarded to the Web Server on the local loopback IP address (127.0.0.1) on port 8080 to which the server is accepting incoming requests.
Figure 4.12 (28): The activities during the execution of private and public services.

In both types of services, the result of the service call travels back to the client-side peer through the reverse path. When it reaches the client-side EA it sent to the client stubs through the loopback IP address and it is eventually forwarded back to the service implementation for further processing.
Public services, SOAP-based or REST-style are executed with direct HTTP calls to the publicly available web server that hosts them. The activity diagram in Figure 4.12 depicts the Execution Phase as described above augmented with the execution of the public services required in the composition.
Chapter 5

p2pSOA Evaluation

In this chapter we evaluate the p2pSOA design and implementation along two axes. First we revisit the requirements that we set forth in Chapter 3 for a new architecture, and we demonstrate how the p2pSOA has satisfied them through the demonstration of applications that access the functionality of the completed product. Second we provide a performance analysis of the middleware layer during the discovery and execution of composite services.

5.1. Requirements for the new middleware and the p2pSOA architecture

In Section 3.2 we defined a set of requirements for a new middleware through a set of motivating applications. A summary list of these requirements is shown below:

1. Pervasive Connectivity: Services of personal devices residing in NAT networks can be accessed from outside their network boundaries

2. Mobile devices should expose services to the trusted group
3. Compose private services with public services from the open Web.

4. Information is only shared between peers in the trusted group preserving user-content privacy.

5. Capability to create and join trusted groups using authenticated mechanisms.

6. The service provider and peers can offer value-added community services through the Cloud.

We revisit these requirements in the description of the each application implementation and we identify which of them are satisfied in each application.

5.2. Use Cases Realization

In order to validate our model and demonstrate that the above requirements are satisfied through our design, we implemented the applications that motivated the design of the p2pSOA architecture. We briefly revisit the functional aspects of the application descriptions that were first introduced in Chapter 3 and then describe the scenarios we chose to implement.

5.2.1. Snap-A-Picture: Control the mobile phone’s camera remotely

In the first application, the actor is visiting a relative. Using his relative’s phone (or laptop), he sends a notification to his wife’s phone that he’s ready to show the relative a stream of pictures of his baby in his crib. She places her phone over the crib. The actor sends a request to snap a
picture of the baby and the picture is displayed on the relative’s mobile device or laptop/desktop. With this application we are exposing the capabilities of p2pSOA to deal with networks that implement NAT and Firewalls, we show the capability to compose private services across these networks and we demonstrate the capability to interface embedded device services over Wi-Fi networks.

**Application Specifications**

The actor visiting a relative’s home opens the p2pSOA Client HTML (C-HTML) page using the p2pSOAWebV application on the relative’s mobile device (or opens the C-HTML page using a browser on the relative’s laptop). The C-HTML page displays a Discovery button. The user presses the discover button and a discovery message is sent to the remote phone through the p2pSOA middleware. The discovery returns the WSDL file of the `snapPic` web service. With the WSDL file, the p2pSOA’s RA generates an AtomServ stub. As explained in Chapter 4, the AtomServ is a JavaScript/Ajax template that is generated automatically by p2pSOA. It includes the calls to the services through JavaScript/Ajax functions. When the actor executes the composite service, a preview of live video is displayed on the remote phone and the `snapPic` web service is invoked in a loop with predefined initial delay (ex. 3 seconds). At the end of the initial delay pictures are snapped and are transferred to and displayed on the relative’s cell phone or laptop. Once done the actor can press the END button that sends a request to the remote phone to stop the live camera preview.
Implementation

The network diagram for this Use Case is shown in Figure 5.1. Two Android phones are involved in this Use Scenario. Both phones are physically located in the same residential home and they are placed side-by-side. However, Phone 1 is connected to the home’s Wi-Fi network 1 while Phone 2 is connected to a neighboring home’s network 2 as shown in Figure 5.2. The JXTA Rendezvous/Relay (RDV/RLY) peer is launched at a Linux node with a public IP address at the Northeastern University campus network. The p2pSOA software is launched on each phone. The software launches a JXTA Edge peer on each phone as well as the Discovery (DA) and Execution (EA) agents, and the Runtime Adapter (RA).

The scenario begins when the p2pSOAWebV application is launched on Phone 2. As mentioned earlier, the p2pSOAWebV application is a WebView (a mobile browser) application that is capable of executing JavaScript and Ajax scripts. The p2pSOAWebV opens to the Discover page and it displays the Discovery button. The user presses the Discovery button and a
Figure 5.2 (30): The server-side phone (left) and the client-phone connected to two separate home networks.

discovery request is initiated to the peer on Phone 1. Once the WSDL file for the snapPic Web Service arrives to the peer on Phone 2, the stubs are generated and the Execution button appears. In addition for this particular Use Scenario, a STOP button is also displayed. This is application specific in order to allow the client to keep receiving images from the server until the STOP button is pressed. When the user presses the Execution button the composite service is executed. The server phone (Phone 1) in Figure 5.3 enters the preview mode for 3 seconds, a picture is snapped and is returned to the client phone. The loop is repeated until the STOP button is pressed at which time the client instructs the snapPic Web Service to exit the preview mode and return to the phone’s default screen.

This application demonstrates that the p2pSOA deployed on the two mobile peers solve the issue of pervasive connectivity and connect across distinct networks (Req.1 and 2). They form
and join a JXTA group during launching and thus they also satisfied Req. 5. Finally, they keep the data local to the devices which satisfies Req. 4.

5.2.2. Family Errands: Organize the grocery shopping between family members

In this application a family member (actor) needs to track the location of the rest of the family members (who are traveling home from work in order to plan the dinner preparation. The family member closer to the supermarket is notified to buy what is needed (shopping list) for the preparations. With this application we demonstrate the capability of p2pSOA to combine private (locations of peers) and public servers (Google Maps API service) services in the same composition and across both mobile devices and personal computers.
Application Specifications

The actor at home or any of the actors on the move, launches the p2pSOAWebV using his/hers phone (or a laptop) that loads the local p2pSOA Client HTML (C-HTML) page. The Client HTML page displays a Discovery button. The user presses the discover button and a discovery message is sent to all the peers in the group. The discovery returns the WSDL files of each of famErrands web services. The RA generates the stubs that are used to invoke all the appropriate services.

Once the stubs are generated the C-HTML page will draw an Execution button. When the actor who initiated the discovery (at home or on the go) presses the Execute button, the composite service is executed sending a call to each famErrands web service at all peers. Each peer returns its location as a [latitude, longitude] pair.

The client composite service uses the coordinate pairs to determine the proximity of each peer to a grocery store. The proximity is determined by sending the location coordinates to Google Maps Places API. Once the peer closest to the desired store is identified a notification is sent to it. Once the notification is accepted by the user’s peer, a map that shows the peer’s current location and the location of the store is displayed on the peer’s cell phone. In addition, the list of the elements that he/she can get from that store, are displayed in a window at the top of the screen. The list is also saved on the SD card of the phone for future reference.

The notification occurs through the famErrandsNotif web service running on all peers as part of a web service bundle (i.e. downloadable WS bundles to support a composite service). When the composite service client calls this web service, a message will be posted to the server peer, which in turn responds
with a confirmation message to the client. If needed in the business logic of the server, a notification flag can also be raised. The exact logic of the `famErrandsNotif` service is generally application specific and will need to be available to all peers running this application when the application is instantiated.
Implementation

The network diagram for this application is shown in Figure 5.4. The Family Member 2’s phone (running Peer 2) and the Family Member 3’s phone (running Peer 3) are placed in different locations around the Northeastern University (NU) campus and are connected to the school’s WiFi. The Family Member 1’s phone (running Peer 1) is placed at a building adjacent to the on-campus grocery store. The CS client is running on a Windows XP laptop that is also placed in the vicinity of Phone 1. The JXTA Rendezvous/Relay (RDV/RLY) peer is launched at NU on a Linux node with a public IP address. The p2pSOA software is launched on each phone and on the laptop client. The software launches the JXTA Edge peers on each phone as well as the Discovery (DA) and Execution (EA) agents, and the Runtime Adapter (RA).

After the discovery phase is completed and the stubs are generated, the execution phase is launched and EA retrieves the coordinates of the three phones and returns them to the CS. The CS sends the coordinates to the Google Maps and Places APIs and it receives and displays a map that shows the locations of the closest ten stores to the three phones as shown in Figure 5.5. We limited the number of stores to ten for clarity even though the maximum number of points of interest that Google Places can return is 25.

The CS also receives a distance matrix with all the distances between the stores and the three phones. It uses these distances to calculate which of the phones is closest to which store and it sends that phone a notification along with the coordinates to the closest store. The notification is shown in the left pane of Figure 5.6. The user is given an option to accept or reject the request. If he/she rejects it then the execution is aborted and the client is notified accordingly. However, if
the user accepts it, the phone displays a map that marks its location and the location of the closest store along with the grocery list as shown in the right pane of Figure 5.6.

This application demonstrates that the p2pSOA deployed on the two mobile peers solve the issue of pervasive connectivity (Req.1 and 2). They also form and join a JXTA group during launching and thus they also satisfied Req. 5. And they also keep the data local to the devices which satisfies Req. 4. Finally they combine services from the public domain (Google Maps and Places) with the private services in the composition, which satisfies Req. 3.

5.2.3. Cloud and Peer-to-Peer Overlay Integration: Discovery services through the Cloud

With this application we demonstrate the capability of p2pSOA to augment the Cloud with the discovery of services as described in Req. 6. We imposed this Use Scenario on the Family Errands application. A peer in the trusted group does not have a web service running locally on the mobile device. The p2pSOA middleware looks for the service in the peer’s Dropbox directory and if it does not find it there it looks in the operator’s Dropbox directory.

Application Specifications

During the Discovery phase, a service is not found in Peer 1. The DA of Peer 1 returns a NOTFOUND flag and the peer’s Cloud ID to the P2PA of Peer 1. If the peer does not have a cloud ID then the peer returns null. The P2PA adds this data in the placeholder for that WSDL document and it sends it to the client. When the Discovery phase is complete, the P2PA on the client side scans the WSDL placeholder for
Figure 5.5 (33): The Google Map canvas on the client showing the closest grocery stores (red markers). The orange circle pinpoints the location of the client.

the missing WSDL document. For each missing WSDL document it uses the corresponding cloud ID to look for links to the web services in the corresponding peer’s Dropbox account. The convention is to use a file called myServices in the Public directory of the user to include all known links to WSDL documents. If the peer cloud ID is null then the p2pSOA middleware looks directly into the provider’s Dropbox account for the allServices file. If the WSDL file is not defined in there either then the Discovery phase is aborted and the composite service stubs are not created.
Figure 5.6 (34): The phone closest to the store receives a notification that a grocery list is available for viewing (leftmost image). If the list is accepted then the map of the phone location and the location of the closest store are displayed on the phone.

**Implementation**

We utilized the same Use Scenario that was used in Section 5.2.2 in order to validate this application. In the scenario, Peer 1 located close to the campus store does not have the famErrands service running locally. When the Discovery phase is executed, p2pSOA determines that the peer does not have the service and it therefore uses Peer 1’s cloud ID to search the user’s Dropbox directory. In this first Use Scenario, the WSDL document is found in the user’s myServices document. The DA retrieves it and places it in the application’s directory.

In a second scenario, the WSDL file is not found in the user’s myServices file in Dropbox. The p2pSOA middleware uses the Service Provider’s default Cloud ID to search the allServices
file in the provider’s directory. The file is found there and it is placed in the application’s directory.

Finally, we tested a scenario where the WSDL file for the famErrands was not found in either directory. In this case the DA notifies the user and it aborts.

With this application we demonstrated that the middleware meets the requirements of involving the Cloud in the discovery of value-added services offered by the Service Provider.

5.2.4. **Book Search: Sharing of book lists between friends**

We implemented a Book Search application in order to demonstrate the capability of p2pSOA to compose SOAP services. The network setup on which we implemented this scenario and conducted measurements is shown in Figure 5.7.
Application Specifications

Three friends decide to create a collaboration group to share access to their books. They run on their PCs a Books service that keeps a list of books available at their home. Friend A wants to look for a book, so she launches the BookSearch composite service with the desired book title as input. The CS first searches in A’s home database and if the book is not found, it searches the remote book databases of her two friends B and C, by contacting the corresponding private services running on their devices. If the book is still not found, the CS contacts the public Amazon.com Product Web Service (APWS) to get pricing information about the book.

Implementation

We implemented this scenario using OSGi containers and bundles. Both the client-side CS as well as the server-side web services were implemented in Java and compiled as OSGi bundles. The Books Web service is deployed in the Axis server bundle running in the OSGi container. We ran all three peers on personal computers running Linux. During discovery the SOAP-based APWS is contacted in order to retrieve from it a WSDL file to help generate appropriate stubs for invoking it.

During execution the CS contacts all peers and retrieves their local book lists. For the cases that the book was found the user was notified. Otherwise the APWS was invoked using the book title as a query string. The price of the book is returned to the user.

We used this application to profile the performance of the p2pSOA middleware based on the user experience. We show the results of the analysis in the next section.
Figure 5.8 (36): Network setup for benchmarking the standard p2pSOA reference implementation.

5.3. Performance Analysis

To evaluate the performance of the p2pSOA middleware, we conducted experiments to compare the discovery and execution phases of the standard p2pSOA implementation with direct HTTP calls to the remote SOAP server in the Book Search application. Since direct HTTP calls cannot traverse NATs, in order to be able to compare the two mechanisms, we executed the server-side peer and the Rendezvous / Relay peer in the same 100 Mbits/sec LAN as the client-side peer, as shown in the Figure 5.8. All peers were executed on Dell Precision 380 desktops running Linux.

To compare the delays in the discovery phase, we measured the Round-Trip Time (RTT) for sending WSDL requests to retrieve the WSDL file of a remote service, first using JXTA and our middleware and then using direct HTTP/GET to the remote Axis engine. A single experiment run consists of 1000 consecutive synchronous WSDL requests that are routed to the server-side peer and the RTT for each request is recorded. In addition we measured the delays introduced individually by each of the p2pSOA layers on the client and server sides. The results are shown in Table 5.1. As depicted in the table, the mean RTT delay introduced by the p2pSOA layers of 172 msecs is somewhat larger than its HTTP/GET counterpart 108 msecs. We attribute this delay
Table 5.1 (2): Comparison of p2pSOA with direct HTTP.

<table>
<thead>
<tr>
<th>P2PSOA OPERATION</th>
<th>MEAN RTT (MSEC)</th>
<th>STANDARD DEVIATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery Phase</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2pSOA client-side</td>
<td>172</td>
<td>31.78</td>
</tr>
<tr>
<td>p2pSOA server-side</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>Axis WSDL response</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Communications</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>Execution Phase</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p2pSOA client-side</td>
<td>76</td>
<td>15.95</td>
</tr>
<tr>
<td>p2pSOA server-side</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Axis SOAP response</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>Communications</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>HTTP OPERATION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP/GET of WSDL (discovery)</td>
<td>108</td>
<td>20.27</td>
</tr>
<tr>
<td>HTTP/POST SOAP (execution)</td>
<td>35</td>
<td>16.89</td>
</tr>
</tbody>
</table>

To the server-side P2PA editing the WSDL file and to the client-side P2PA updating the MRT (see Section 4.7.2). However, from the user experience perspective, this delay is still small considering that the discovery of resources typically takes a considerable amount of time.

To compare the delays in the execution time, we measured the RTT for sending SOAP calls from the CS bundle to a remote service with our middleware and with direct HTTP/POST. Table 5.1 summarizes the results from 1000 SOAP calls sent by the client. Overall, we observed mean RTT of 76 msecs that is slower but comparable to the transport of SOAP messages over direct HTTP/POST, which was 35 msecs. As it can be seen from the breakdown, most of the delay occurs on the server-side where the P2PA edits the SOAP messages.

In the Book Search application experiment after initialization, the CS application iterates 50 times through the Discovery and Execution phases. For each iteration we recorded the duration of each phase, including the time it takes for the Stub Generation operation to complete. The average times of the experiment are provided in Table 5.2. The WSDL retrieval times and the
Table 5.2 (3): Metrics for the Book Search application

<table>
<thead>
<tr>
<th>BOOK SEARCH APPLICATION</th>
<th>MEAN TIME (MSEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discovery Phase</td>
<td>854</td>
</tr>
<tr>
<td>WSDL retrieval RTT from Service B</td>
<td>250</td>
</tr>
<tr>
<td>WSDL retrieval RTT from Service C</td>
<td>275</td>
</tr>
<tr>
<td>WSDL retrieval RTT from Amazon</td>
<td>329</td>
</tr>
<tr>
<td>Stub Generation</td>
<td>351</td>
</tr>
<tr>
<td>Execution Phase</td>
<td>470</td>
</tr>
<tr>
<td>SOAP call RTT to Service B</td>
<td>125</td>
</tr>
<tr>
<td>SOAP call RTT to Service C</td>
<td>111</td>
</tr>
<tr>
<td>SOAP call RTT to Amazon.com</td>
<td>234</td>
</tr>
</tbody>
</table>

SOAP calls for Services B and C are slower than the ones reported in Table 5.1. This is due to JXTA’s connectivity middleware, specifically the Relay peer, for traversing firewalls and dealing with NAT. Overall, we believe that the discovery and execution delays are reasonable and acceptable from the user’s point of view.

Finally, we measured the amount of memory (RAM) that the p2pSOA requires. We measured the executable sizes and the memory footprints of the standard p2pSOA components. The former indicates permanent storage and the latter RAM requirements. In these measurements we included requirements by the 3rd party components that we utilize, namely the Axis Engine WSDL2Java tool. Our findings are summarized in Table 5.3. The overall RAM size of our middleware is below 2 MB. Based on this results is what motivated us to implement the Mobile-p2pSOA version of the design verifying that the middleware is well suited for mobile phones.
Table 5.3 (4): Memory footprints of the standard p2pSOA

<table>
<thead>
<tr>
<th>SOFTWARE MODULE</th>
<th>EXEC SIZE (KB)</th>
<th>RAM (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>End-User Configuration Panel</td>
<td>7.1</td>
<td>61.1</td>
</tr>
<tr>
<td>Discovery Agent (DA)</td>
<td>42.0</td>
<td>284.3</td>
</tr>
<tr>
<td>Execution Agent (EA)</td>
<td>49.0</td>
<td>189.5</td>
</tr>
<tr>
<td>Runtime Adapter (RA)</td>
<td>29.6</td>
<td>76.3</td>
</tr>
<tr>
<td>WSDL2Java</td>
<td>6.9</td>
<td>560.6</td>
</tr>
<tr>
<td>P2P Adapter (P2PA)</td>
<td>149.0</td>
<td>240.5</td>
</tr>
<tr>
<td><strong>Standard p2pSOA Suite Totals</strong></td>
<td><strong>283.6</strong></td>
<td><strong>1412.3</strong></td>
</tr>
</tbody>
</table>
Chapter 6

Summary and Conclusions

With this work we defined, specified, designed, implemented, and validated p2pSOA, a new middleware architecture that enables the collaboration of trusted peers through service composition. It seamlessly offers to the stakeholders of the service compositions the capability to access and execute in a privacy preserving distributed manner each other’s personal services. In addition it allows them to combine personal private services with publicly available internet services into complex composite services with enhanced capabilities. These capabilities enable the creation of ad-hoc Peer-to-Peer (P2P) communities that promote collaboration between trusted peers through distributed service and content sharing. With p2pSOA mobile users may contribute services hosted on their devices to the community or request and consume services that the community offers. Finally in the p2pSOA based collaboration model, service providers can offer value-added community services through existing Cloud technologies.

The main contributions of this research are:

- The p2pSOA middleware architecture combines the benefits of the P2P computing with the heterogeneity aspect of the SOA paradigm and offers:
  - naming and addressing of devices that reside in networks that implement NAT without the need of service-level proxies or intermediaries.
the participation of mobile devices in trusted communities and collaboration with other members through service composition.

- the sharing of services and content among trusted peers through the distributed execution in a true P2P fashion of composite services which combine private and public services.

- the integration of the P2P model with the Cloud model that enables service providers to offer community services to trusted groups through the Cloud.

- privacy of user content by keeping the user information only on personal devices. It utilizes the concept of trusted group in which peers can trust the services offered by members of the group.

⇒ We have developed a working p2pSOA prototype that:

- adopts the Web Services execution model. Utilizes scripting on mobile devices to allow the automatic generation of client stubs at runtime without the need for compilation.

- supports services hosted on both mobile devices and personal computers

- combines private (device-level) and public (internet) services into composite services

- utilizes JXTA as a secure P2P overlay.

- integrates Cloud capabilities for discovering services

⇒ We do not modify the 3rd party software incorporated into the prototype but rather use interfaces (adapters) that can be substituted to create different p2pSOA instances.
Different types of transport and distributed services can thus be accommodated making p2pOSA agnostic to their runtime.

Both the standard and the mobile implementations of p2pSOA pave the road for future work in the area of service composition over peer-to-peer overlay networks. Immediate extensions and improvements to the work include:

- Integration of p2pSOA with social networks, such as Facebook, to enable direct execution of composite services from a Facebook page.
- Implementation of semantic discovery and matching for finding missing services in a set of public services.
- A mechanism that accepts user-configurable criteria for choosing an appropriate WSDL file from a set of WSDL files that match the same query.

There are also a number of far-reaching improvements and directions that we envision for the future. These include:

- Improvement of the discovery mechanism to make use of the peer cache offered by the P2P technologies.
- Enhancement of the User Interface to include additional functionality. One such enhancement may be the capability to terminate the execution of composite service at any time.
- Design and implementation of a Usability Analysis model: Although we offered a set of metrics to measure how well p2pSOA matches the requirements and how it performs during the various Use Cases and Scenarios, we believe that such an analysis can help us further gain an insight as to how the product enhances the user experience.
Integration of a semantic workflow tool with the p2pSOA front-end for enabling the user to define the composition semantically.

Extension of the middleware layer to include a Web Services Adapter (WSA). A WSA can adapt to different Web Services technologies enabling the support of different service implementations by the middleware.
Bibliography


