A P2P SOA Enabling Group Collaboration through Service Composition

Demetris G. Galatopoullos
Electrical and Computer Engineering Department,
Northeastern University,
Boston, MA 02115, USA,
demetris@ece.neu.edu

Dimitris N. Kalofonos
Pervasive Computing Group,
Nokia Research Center Cambridge,
Cambridge, MA 02142, USA,
dimitris.kalofonos@nokia.com

Elias S. Manolakos
Department of Informatics and Telecommunications,
University of Athens,
Illisia, Athens 15784, Greece,
eliasm@di.uoa.gr

ABSTRACT
The Service-Oriented Architecture (SOA) paradigm was introduced for exposing business processes as services and enabling their interaction and composition over the Internet. The same computing model can potentially be extended to services of personal devices. As personal devices become network-aware their services can be made available (by their owners) to members of trusted peer groups, thus allowing them to compose new distributed collaborative applications. However, dealing with firewall traversals, NATs, mobility and issues of service-level naming and addressing stand in the way of this vision. In this paper we introduce a P2P SOA middleware architecture that addresses such problems of pervasive connectivity without requiring any intermediaries or changes to the service implementations. We present the basic elements of the architecture and the design of a specific instance of it, which enables the P2P service discovery and execution of composite personal services in distributed OSGi containers over JXTA.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems —Distributed applications
D.2.11 [Software Engineering]: Software Architectures.

General Terms
Design.

Keywords
Peer-to-peer service oriented architectures, pervasive connectivity, web services.

1. INTRODUCTION

Personal devices, such as mobile phones, digital music players and digital cameras are commonplace in our lives. As these devices become increasingly network-aware, it becomes technically feasible to combine services from different devices and deliver to users, groupware [10] capabilities and an enhanced user experience. 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; a family relative overseas could share our home web camera. But how can non-expert users reap the benefits distributed composite services have to offer? The distributed computing model known as Service-Oriented Architecture (SOA) [20] is the emerging paradigm for organizations to expose and compose their heterogeneous services over the Internet. Therefore, it seems to be a natural candidate for also allowing personal devices to expose their capabilities in trusted groups of peers and facilitate their distributed composition and execution.

Before we proceed, let us define some terms to be used in this paper. We consider two types of elementary services: public and private. Public are the services which are available in the public domain, e.g. Web Services (WS) accessible over the Internet. Private are services offered by privately-owned personal devices which do not provide the infrastructure to expose them globally. Our Composite Services (CS) may combine both private and public elementary services into more complex tasks. Collaboration, in our context, means that members of a trusted group contribute services voluntarily so that other group members can utilize them with their own services in composite services, which are executing in a distributed fashion. Finally, the terms server-side and client-side are used only to specify which peer is the service producer or the service consumer. In our P2P architecture, each peer can play both roles and be both a producer and a consumer of services.

1.1. Research Motivation

Composing private services presents a set of unique challenges not present in standard web service composition. One of the important challenges is providing *pervasive service connectivity,* i.e. dealing seamlessly with firewall traversals, NATs, mobility (IP changes) and issues of service-level naming and addressing. The problems of traversing firewalls and dealing with NATs have been considered by the WS-Dispatcher (WSD) [22], the TARGET [13] two-way SOAP router, and the IBM Web Service Gateway (WSG) [6]. However, all these projects either utilize intermediary proxies and centralized service registries or the WS-Addressing Proposal [24]. On one hand, the problem with intermediaries at

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICPS'08, July 6–10, 2008, Sorrento, Italy.
Copyright 2008 ACM 978-1-60558-135-4/08/07...$5.00.

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 solves the problem of pervasive service connectivity without imposing any modifications to the service implementations.

1.2. Research Problem

Our main objective is to define and develop a middleware architecture that can solve the pervasive service connectivity problem in order to incorporate in the SOA model private services that peers are willing to share. These services may operate on personal devices that are either constrained by operators' firewalls (e.g. mobile phones) or confined within a home network.

The main contributions of this paper and of the *p2pSOA* architecture we are introducing are summarized below:
1. We extend the P2P paradigm all the way up to the service level by adding a middleware layer that adopts the Web Services execution model and allows us to intercept, modify on the fly as needed, and route SOAP messages end-to-end. We can thus deal with the naming and addressing of devices that are behind NATs/Firewalls and achieve the distributed execution of services composed of private and public services in a true peer-to-peer fashion. We show how this can be accomplished without the need of any service-level intermediaries.
2. We demonstrate the *p2pSOA* architecture capabilities through a prototype of a particular instance that we have built. Specifically the instance *p2pSOA-OJ* uses OSGi [19] as the service container and runtime environment and JXTA [9] as the P2P overlay. We emphasize here that we do not modify in any way OSGi or JXTA, but rather use interfaces (adapters) that can be substituted, if desired, to create a different *p2pSOA* instance.

1.3. Related Work

There are a number of existing solutions that introduce the P2P paradigm to the service level through a middleware architecture. The Wings middleware [15] 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. Our middleware shares a similar design of adapters for interfacing P2P overlays. The design in [5] also defines a middleware architecture for separating the P2P overlay network from the application and thus enabling the utilization of different P2P overlays. The p2pSOA middleware separates in a similar manner the P2P details from the service implementation by not imposing APIs on the service implementation. Similarly, the UbiServInt [4] introduces a layered infrastructure for the P2P service composition of services on mobile devices. Its main focus is fault tolerance and context awareness offered to mobile wireless devices. Our focus is different from the above three designs. We take advantage of the pervasive connectivity offered by P2P overlays and we generate client-side composite services composed of private and public services that we execute over the P2P overlay. Two last designs in this category are the Self Serv environment [2] that presents a layered middleware architecture for provisioning Web Services over a P2P network without a centralized controller, and the p2pWeb project [17] 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 URLs through a proprietary Naming Service. Unlike p2pSOA, these designs do not utilize P2P overlays for accessing and composing private services but instead they use them for either composite service orchestration for enhanced execution scalability or for simple discovery and availability of services through Web browsers.

Next, we describe two designs that although they do not define a middleware layer as the aforementioned projects, they do achieve the integration of P2P overlays with SOA. First is the NEMO [3] service orchestration framework that utilizes 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 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. The second design [23] is a service composition architecture provided on top of the SOA model that manages devices and their services and provides algorithms for their semantic and dynamic composition. Although we share the same goals of providing the user with the capability to compose services from personal devices without imposing changes to the service implementation, p2pSOA extends the local network model to deal with distributed services and their execution over P2P overlays.

Finally, there is also a set of solutions that, similarly to our p2pSOA-OJ architecture instance, combine OSGi and JXTA in order to provide service sharing between containers. One example is the project P2PComp [7] that enables OSGi bundles to access remote services offered by remote OSGi containers through port-like abstractions [16] at the container level. However, it imposes implementation requirements to the client-side bundle in order to access this capability. Our design utilizes interception techniques to capture and manipulate messages from services without requiring an API. Another example is the VHE project [14] that extends OSGi with system bundles that allow the access of devices across smart homes. However, it does not deal with the composition of services executing on distributed devices.

The rest of the paper is organized as follows: Section 2 presents use cases that demonstrate the requirements and motivate our architecture; Section 3 outlines the main elements of the proposed SOA architecture; Section 4 presents an instance of *p2pSOA* combining specifically OSGi and JXTA; Section 5, outlines our implementation and experience with its proof-of-concept prototype; we finally summarize our findings and point to future directions in Section 6.

2. USE CASES AND REQUIREMENTS

We present and analyze here two use cases (UC) that motivate our work. In UC1, depicted in Figure 1, George wants to find out if any groceries are needed at home on the way back from a friend's place. He and his wife maintain a list in a `GroceryList` service and can add items to it through a Display Terminal in their kitchen.

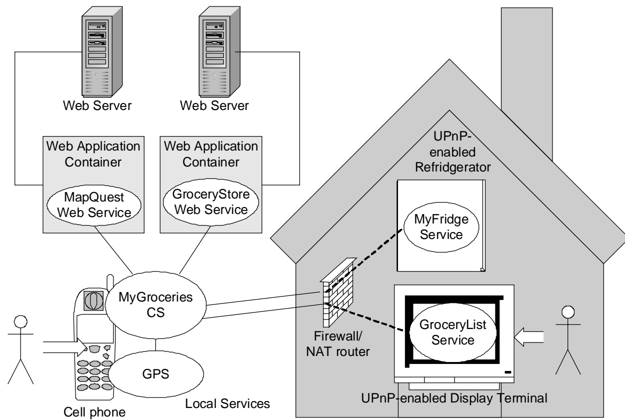

Figure 1: Motivating use case UC1.

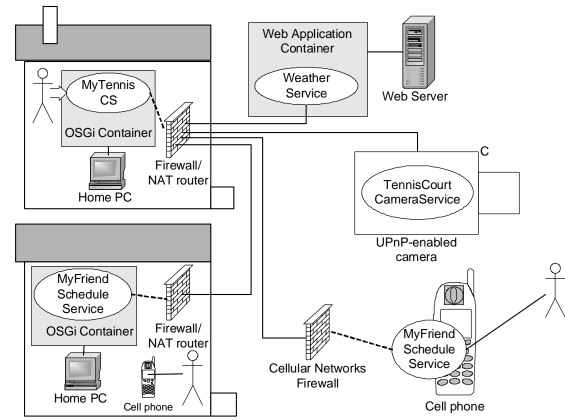

Figure 2: Motivating use case UC2.

Their smart Refrigerator also provides a service (MyFridge) that keeps track of its contents. George can launch on his mobile phone a simple MyGroceries task he has composed (CS) that can find the stores that are close to his current location, carry the items needed and offer the lowest price. In UC2, depicted in Figure 2, three friends living in an apartment complex, like to play tennis at the tennis courts located at the other end of the complex. One of the friends, Lori, wants to invite a friend for a tennis match. She wants to find out if any of the tennis courts are currently available, and if so, she also needs to find out if the weather will be clear over the next couple of hours. Lori can launch the MyTennis CS that she has composed on her home PC which can find if any of her friends are available and send one of them an invitation to his/her mobile phone.

Approaching UC1 from the system's perspective, we identify the following elements. A CS executing on George's phone contacts the MyFridge service and the GroceryList service executing on two UPnP-enabled devices in his home network. Both services return the grocery lists they maintain and from these the CS calculates the list of missing items. The CS accesses a local GPS service running on George's cell phone to get his current location. Then the CS sends the location to the MapQuest web service that returns a list of grocery stores within a certain mile radius (chosen by George) along with directions to each one of them. The CS then contacts the GroceryStore web service of each of these stores, using the desired grocery items as parameters and retrieves availability and prices. George chooses the best store based on distance and/or pricing and the directions to it are displayed on his cell phone.

In UC2, Lori launches the MyTennis CS in her home PC that first contacts the UPnP-enabled TennisCourtCamera service running in the camera overlooking the courts. The court images are displayed on Lori's PC. After reviewing the images, if the courts are free Lori accepts the next action. The CS then contacts her preferred Weather forecasting WS which returns the weather prediction for the afternoon. If the weather is expected to cooperate, she accepts the next action that prompts the CS to contact her friends' MyFriendSchedule private services for their availability. One of her friends is running this service on her home PC whereas the other one on her cell phone. Once the schedules are retrieved and if at least one friend is available, Lori makes a selection and the CS initiates an invitation by sending a message to the friend's mobile phone.

Both use cases expose a set of challenges from which we derive the following requirements. First, accessing the services of personal devices from outside their local network requires solving the problem of pervasive service connectivity which includes home firewalls/NATs, addressing and mobility. As it can be seen, both clients and servers may be mobile devices. Therefore, the client should be able to connect to the server regardless of his/her location. Second, UPnP-enabled devices, such as the tennis court camera, expose their services only in the network they reside. There needs to be a mechanism to make their services available for discovery and composition with public services across distributed peers. The issue of trust is also of great importance. The three friends want to share their schedules only within their trusted group and therefore they should be able to form and join this group requiring membership authentication. In the next section we present our proposal for a service oriented P2P architecture that can meet these requirements.

3. THE p2pSOA ARCHITECTURE

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 order to address these requirements we propose here a four-layer architecture, depicted in Figure 3. We provide a high-level overview of each layer starting from the bottom.

The lowest layer of the architecture is the Distributed Services layer. At this layer we have services provided by different hosts. If the service is public then the provider of the service is directly accessible through the Internet. However, if the provider is a peer in a trusted group, then we employ our p2pSOA middleware at the provider side and a P2P overlay to expose the service as a WS and address the pervasive service connectivity issues between the consumer and the provider. Finally, if the provider is the local host there is no need for a connectivity layer. The middleware layer exposes them as Web Services enabling their availability.

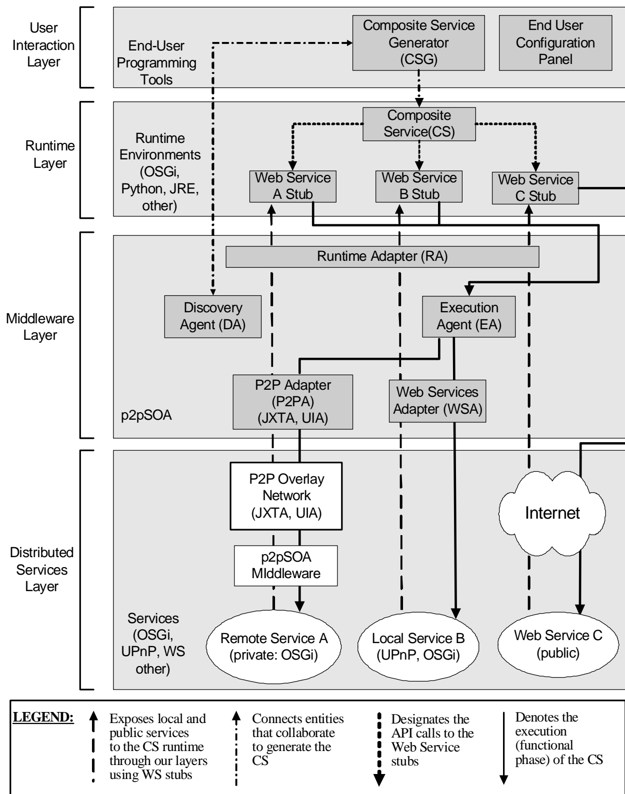

Figure 3: Overview of the *p2pSOA* architecture.

The next layer up is the *p2pSOA* Middleware layer that includes adapters and agents. Adapters offer a level of indirection with the entities they adapt to. For example, in order for local-host services from the Services layer to be involved in service composition, they are adapted to become Web Services. Adapters also separate the various application concerns, for example the peer-to-peer communications code, from the service implementation. For example any service can be adapted to execute on a variety of CS runtimes (e.g. OSGi, Python, JRE). Another example of an adapter is the P2P Adapter that allows us to incorporate different P2P overlay technologies (e.g JXTA, UIA [8]) without modifying the core of the middleware. For instance since JXTA's rendezvous routing scheme does not exhibit optimal scalability [1], it may be necessary to replace it with another overlay for certain applications. 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. Subsequently, the Execution Agent's responsibility is to route the calls to these services through the overlay without requiring any modifications to the CS.

The next layer up is the Runtime layer. We choose to include a runtime layer because it allows the CS to be platform independent. Since p2pSOA adopts the WS model of execution that utilizes client stubs, we need to be able to utilize these stubs and generate the CS that calls them for different CS runtimes, including mobile device platforms.

(a) Books.wsdl

```
...
<wsdl:definitions targetNamespace=
    "http://192.168.1.2:8080/axis/services/remoteBooks"
    ...
  <wsdl:portType name="BooksImpl">
    <wsdl:operation name="getBooks">
    ...
```

(b) Books**5DE8688C9E63479DA1C8CB215F52234A.wsdl**

```
...
<wsdl:definitions targetNamespace=
    "http://127.0.0.1:25766/axis/services/
remoteBooks5DE8688C9E63479DA1C8CB215F52234A"
    ...
  <wsdl:portType name="BooksImpl">
    <wsdl:operation name=
"getBooks5DE8688C9E63479DA1C8CB215F52234A">
    ...
```

Figure 4: (a) WSDL fragment, (b) same fragment after it has been modified by the P2PA.

Finally, the top layer in our architecture is the User Interaction layer. This layer captures the user intent of the CS in a non-grounding representation, such as a workflow diagram or a semantic graph. Our middleware discovers the atomic services described in the representation and provides to the User Interaction layer the necessary service descriptions for automatically generating the CS.

We describe the four layers of the p2pSOA architecture in detail in Sections 3.2-3.5.

3.1. The p2pSOA End-to-End Message Routing

The main goal of our middleware architecture is to enable true peer-to-peer execution of composite services by combining private and public services over P2P overlay networks. The following SOA proposes service invocation via SOAP [21]. The SOAP implementation uses IP addresses and DNS as part of the XML documents that constitute its messages for service addressing. Since hosts may actually have private addresses that are meaningless in a global scale, one of the innovations of our the paper is that we are intercepting and manipulating on the fly these 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. We describe here how we can achieve this.

In our solution, we introduce a mechanism based on a unique fixed-length ID called *Peer Unique ID* (PUID) to help us deal with addressing. Our middleware maintains a routing hash table, called *Message Routing Table* (MRT) that maps the PUIDs to endpoint identifiers understood by the P2P layer. Examples of P2P endpoint identifiers are the Server Pipe ID (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 service that is part of the CS, before the file is delivered to the client-side.

This results in stubs at the client-side that include the PUID in the service method names. In Figure 4(a) we show a fragment of a `Books` service WSDL description before it is modified by the *p2pSOA* middleware and in Figure 4(b) after it is edited. As shown, 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(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 (25766) 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 25766. 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, 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 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.

3.2. The Distributed Services Layer

The Distributed Services layer represents private and public distributed services. For example, in Figure 3, `Service A` may be a remote service offered by an OSGi bundle executing on a remote trusted peer host. This service participates in the CS composition through the *p2pSOA* middleware using the underlying P2P overlay. The P2P overlay transports the discovery requests to retrieve the WSDL for `Service A` from the appropriate peer. At runtime, it routes the SOAP messages from the client-side peers to the `Service A` location. Next, since the services in this layer are distributed and their interfaces are well-known, it is possible to develop adapters that can expose these interfaces as Web Services (WS) interfaces. For example `Service B` is a local service offered by a UPnP device. Its interface is exposed as a WS so that it can be discovered and employed in the CS. Finally, `Service C` is a public Web Service executing on a remote Web Server. `Service C` is discovered using standard discovery methods, e.g. UDDI registries [18].

3.3. The Middleware Layer

The *p2pSOA* middleware layer is the core of our design. It is composed of *adapters* that expose local services as Web Services, deal with the issues of pervasive service connectivity and distributed group management, and 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. Below we describe the major entities of the middleware and their roles.

The Web Services Adapter (WSA): The WSA offers a layer of indirection allowing the use of different distributed services frameworks. The WSA is responsible for generating WSDL files for local services, such as those offered by UPnP-devices or OSGi bundles, in order to employ them as WS in a CS. The adapter generates a WSDL file each time a service is installed in the local system and it subsequently stores it so that it can be accessed during discovery. The WSA is also used for processing SOAP messages, because in general, local services may not be able to process them. Therefore, during the CS execution, the API calls to the service stubs actually make calls to the WSA. The results of these API calls, which are invoked by a local or remote CS, are SOAP messages that the WSA converts to a form that the local services implementations understand.

The P2P Adapter (P2PA): 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 the 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 is 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 WSA and receives the appropriate WSDL file. Subsequently, the server-side P2PA modifies the WSDL file (includes the PUID and modifies the URL as discussed earlier) 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, and then retrieves the PUID of the server-side peer from the WSDL file in order to update the MRT.

During execution, the client-side P2PA intercepts the outgoing SOAP message, retrieves the PUID from it to use it for looking up the corresponding P2P endpoint in the MRT, and forwards the SOAP messages to the server-side peer. The server-side P2PA is then responsible to reverse these SOAP modifications before passing the SOAP requests to the EA.

The Runtime Adapter (RA): The RA offers a layer of indirection allowing the use of different CS runtimes. The RA generates and compiles on the fly the service invocation client stubs for a particular CS runtime using the corresponding WSDL files. The client stubs are classes that include the language bindings and API calls necessary for the client application to call a remote Web Service. Calls to the remote service methods are routed through the client stubs that in turn generate the appropriate SOAP messages that our middleware routes to the remote service implementation. The client stubs process the SOAP 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.

The Discovery Agent (DA): The client-side DA is responsible for accepting the atomic service descriptions submitted by the User Interaction layer, generating discovery queries for them (in our

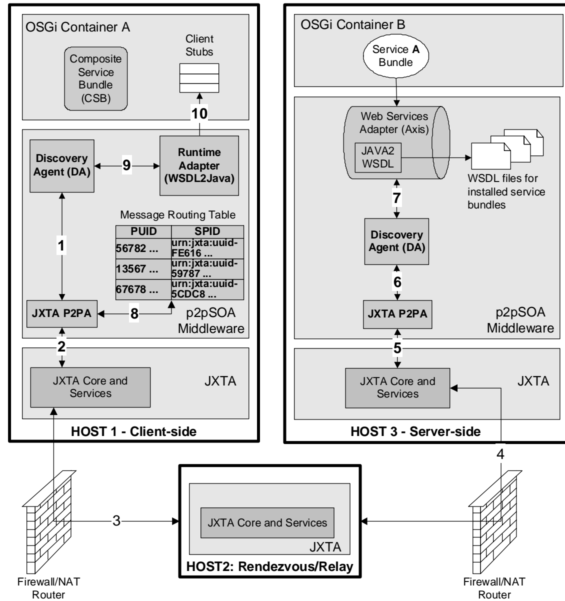

Figure 5: Functional flow of the Discovery Phase.

future plans, these queries will be semantically defined) and forwarding these queries to the P2PA. Queries to public services follow well established mechanisms (e.g. UDDI). When the WSDL files are returned, there is a possibility that multiple WSDL files matching the same query will be returned by different peers. In such cases the DA selects the appropriate WSDL files based on a set of criteria that can be user-configurable, e.g. considering end-to-end delays. 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.

The Execution Agent (EA): On the client-side the EA listens to the localhost interface on the TCP port reserved by the *p2pSOA* middleware (port 25766). Thus, during the CS execution, it intercepts the SOAP calls originating from the stubs and it forwards them to the P2PA that transports them to the server-side peer. The server-side P2PA removes the PUID added to the SOAP message by the client-side stubs and then forwards the SOAP messages up to the server-side EA, which in turn forwards them to the WSA in the form of HTTP requests.

3.4. The Runtime Layer

The next layer in our architecture is the Runtime layer. We introduce this layer for CS independence from the underlying OS and ease of creation of the CS. By abstracting this layer we have the flexibility to generate appropriate code for executing the CS on a different choice of runtime based on a set of criteria, such as speed or memory requirements. With p2pSOA we target a variety of platforms including mobile device environments. For example, composing text scripts on the fly instead of compiling native executables is a task feasible even for constrained devices, such as mobile phones. Moreover, friends that own devices that execute services using the same runtime, can exchange a CS without the need for modifications.

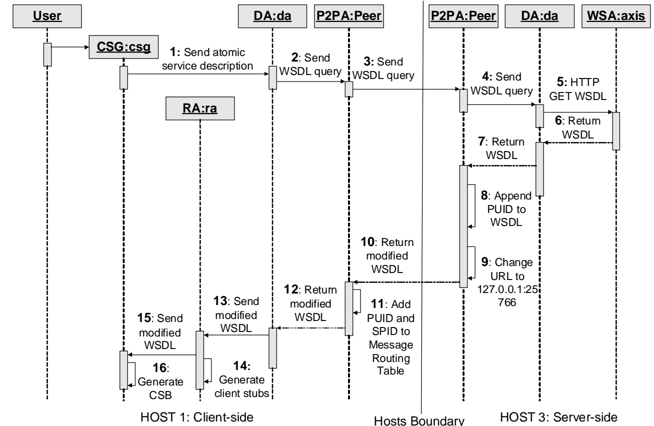

Figure 6: UML sequence diagram corresponding to the Service Discovery Phase.

3.5. The User Interaction Layer

Although this layer is part of on-going research we describe here some preliminary ideas. We expect this layer to employ a set of end-user programming tools [11] for capturing the user intent. These tools provide the user with the capability to semantically describe a CS through a services composition workflow in a non-grounded form such as an XML-based semantic language description. For our purposes, we desire that these tools forward to the DA the set of atomic service descriptions comprising the CS. Once the DA retrieves the WSDLs from the P2P layer in response to its queries, it forwards them to the Composite Service Generator (CSG), which generates a grounded (concrete) form of the CS that can execute in the runtime environment.

Finally, in order to give the user the capability to configure our middleware and its components, e.g. manage the trusted group membership, we introduce an End-User Configuration Panel tool. This tool presents a Graphical User Interface (GUI) that allows the user to launch and terminate the underlying p2pSOA and to enter values for configurable parameters, such as permissions lists for controlling access to local services or IP addresses for the configuration of the overlay network.

4. THE *p2pSOA-OJ* SYSTEM DESIGN

In the rest of the paper, we will concentrate on the description of an instance of the above general p2pSOA in which the services are executing in OSGi containers. We selected OSGi because it is a popular component model for enterprise and residential service oriented applications. For the P2P overlay network we selected JXTA because it 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 [9]. Both OSGi and JXTA are important technologies extensively used by the pervasive computing community. We will refer to this specific instance as *p2pSOA-OJ* (O-OSGi and J-JXTA) for brevity.

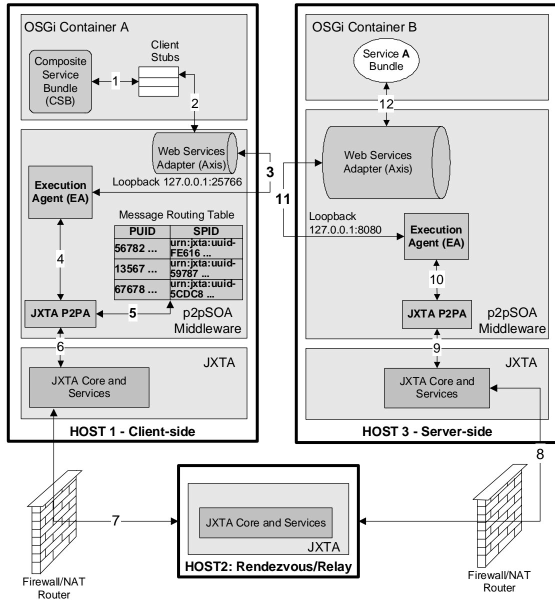

Figure 7: Functional flow of the Execution Phase.

OSGi specifies a component model and a container for services. Services are deployed in it in the form of software *bundles* that can be installed, configured and updated at runtime, without the need to reboot the container. These services can be discovered by other bundles in the local OSGi registry and combined to form more complex services. OSGi can be used both for hosting services, as well as for the CS runtime. In the instance architecture we use the Apache Axis Engine [25] in the role of the WSA. This is because Axis generates WSDL files for OSGi bundles. The Axis is itself a bundle that has been imported into OSGi containers [12]. In the role of the RA we utilize the Axis WSDL2Java tool [25] that generates Web Service stubs from WSDL files. Since the CS is an OSGi bundle itself, we call it Composite Service Bundle (CSB) for this instance. Finally, the OSGi specification provides a methodology for developing bundles that interoperate with UPnP-enabled devices and control points. This enables UPnP devices to advertise their services with the OSGi registry and to discover and utilize services offered by other OSGi bundles.

The P2PA entity in the *p2pSOA-OJ* architecture instance interfaces the middleware layer to the JXTA P2P technology. JXTA peers can form trusted groups in order to share services. This capability fulfills the distributed group management requirement for the P2P overlay in our architecture. Peers can discover other peers using a special type of a JXTA peer called Rendezvous which is responsible for keeping track of peers, groups and other resources in the network. The Relay is another special type of JXTA peer that is responsible for forwarding messages between edge peers residing behind firewalls and NATs. The Rendezvous and Relay peers fulfill the second requirement for the P2P overlay in our architecture, i.e. dealing with pervasive connectivity. Communication in JXTA is offered by *pipes*, a similar concept to the UNIX pipes, which are based on TCP sockets. The primary responsibility for the P2PA is to launch the underlying JXTA platform and route messages over the JXTA

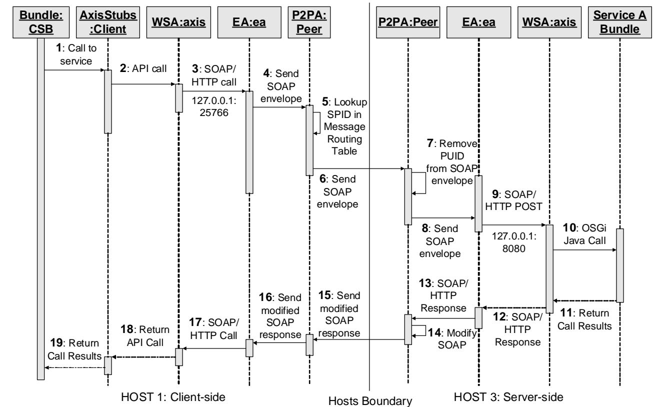

Figure 8: UML sequence diagram for the Execution Phase.

overlay. The P2P endpoint entry in the Message Routing Table (MRT) is the Server Pipe ID (SPID) generated by JXTA for the server pipe of each service producer peer. The P2PA saves the SPID in persistent storage so that the same one is always used for the same pipe. In the rest of this section, we will explain how the *p2pSOA-OJ* architecture works by presenting the functional flows of the three phases of its operation: initialization, service discovery, and service execution.

4.1. Initialization Phase

An example of the entities involved in this phase is depicted in Figure 5. We utilize three hosts: HOST 1 is the client-side host where the discovery requests originate. HOST 3 is the server-side host where the service implementations reside. Both of these hosts execute behind firewall/NAT routers. HOST 2 is the Rendezvous/Relay and it is in the public domain outside any firewalls/NATs. The configuration of the Rendezvous/Relay is done through the End-User Configuration Panel (Figure 3) and its logic is independent of our middleware. Once it is configured, the user may use out-of-band methods (e.g. email, phone call) to send invitations to friends to join the group with instructions, e.g. the IP address of the Rendezvous and the group password.

Upon startup, the Rendezvous creates the group and the peers ask to join it using its name and the authentication credentials. Once granted permission, each of the peers generates a JXTA server pipe (for their services) and client pipes (for accessing services) and advertises them with the Rendezvous. In our example, only the server pipe of the server-side peer is utilized although in reality both peers may generate server pipes. 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. This completes the JXTA initialization.

The next step is for the P2PA to generate the PUID for its peer. At the middleware layer, the DA, EA and RA are launched and they connect to the P2PA using TCP sockets. Similarly, at the runtime layer, the OSGi containers are launched and they activate their bundles. On HOST 3 Service A registers with the WSA (Axis Engine).

4.2. Discovery Phase

The entities involved in the Discovery Phase and the interactions between them are also depicted in Figure 5 and in the UML sequence diagram in Figure 6. Since the steps in Figure 6 show greater detail, their numbers do not match the step numbers in Figure 5. Finally, the Rendezvous/Relay (Host 2) is not shown for clarity in the UML diagram.

In Figure 5 we assume that the CSG has already given the DA the atomic service descriptions of the services in the CSB. Subsequently, the DA in Step 1 of Figure 5, forwards the service queries to the P2PA. In Steps 2-5, the P2PA sends the queries to the server-side peer P2PA. The server-side P2PA forwards the queries to the server-side DA in Step 6. The server-side DA retrieves the WSDL files from the local WSDL repository that the WSA (Axis) has generated. After the WSDL files are returned, the server-side P2PA edits the URLs in them to include the PUID and the 127.0.0.1:25766 IP:port pair (as explained in Section 3.1) before passing them down to the JXTA layers. For simplicity in Figure 5, we do not show the reverse path from the server-side to the client-side after Step 7. Continuing from Step 8 on the client-side, upon receiving the WSDL file, the P2PA enters the PUID along with the SPID corresponding to HOST3 into the MRT table. This SPID can be retrieved from the server pipe's advertisement that the client-side P2PA received from the Rendezvous peer. In Step 9, the WSDL files that are chosen by the DA, based on user-defined configurable criteria, are forwarded to the RA that executes the Axis WSDL2Java tool in Step 10 in order to generate the client stubs. The client-side DA also sends the WSDL files to the CSG which uses them to ground the CS description expressed by the user. The output of the CSG is the CSB.

4.3. Execution Phase

The entities involved in the Execution Phase and the interactions between them are depicted in Figure 7 and in the UML sequence diagram in Figure 8. Similarly to the UML diagram in Figure 6, the steps in Figure 8 are annotated with the corresponding event description and they do not match the step numbers in Figure 7. Also the Rendezvous/Relay (Host 2) is not shown for clarity in Figure 8.

Following Figure 7, once launched by the user, the CSB initiates an execution session by issuing a call to the client stubs (Step 1). Then the WSA (Axis) generates a SOAP/HTTP call destined for the remote service. Because the URL was edited by the remote P2PA to be 127.0.0.1:25766, the SOAP message is intercepted by our EA. Specifically, the client-side EA intercepts the SOAP call and passes it to the client-side P2PA. Because we included the PUID in the WSDL files, and therefore the stubs included it in the outgoing SOAP messages, the P2PA retrieves the PUID from the SOAP message. It looks up the corresponding SPID in the MRT table in Step 5 and then passes the SOAP message to the corresponding server pipe in Step 6. In Steps 7 and 8, the JXTA Relay transports the message across the P2P overlay. Since in our design we used the Axis Engine unmodified as the WSA, the server-side P2PA strips the PUID from the service method name in the SOAP envelope and passes the message to the server-side EA in Step 10. The EA wraps it in a SOAP/HTTP call and then submits it to the WSA in Step 11 as a standard SOAP message

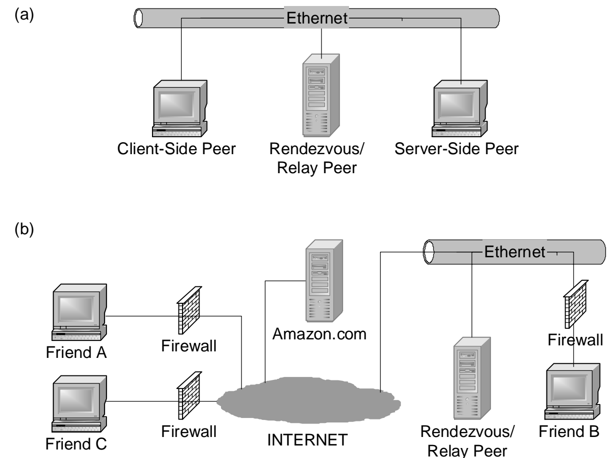

Figure 9: Network setup for (a) benchmarking the p2pSOA, (b) testing the BookSearch application.

over the loopback interface. The WSA processes the message and passes it to the destination service in Step 12. The results of the SOAP service call from the service implementation are sent in the reverse direction on the same path back to the client-side P2PA.

5. PROOF-OF-CONCEPT PROTOTYPE

In this section we present our proof-of-concept prototype implementation that demonstrates the functionality of the *p2pSOA-OJ* instance architecture. We also describe our experience testing the prototype.

The middleware was implemented using the Java 1.5 programming language for platform independence. For OSGi we selected Knopflerfish OSGi Release 4 implementation [12] because it has ported Axis 1.4 as a bundle, offers good support, and provides a GUI for managing bundles. All elementary services in the prototype are OSGi bundles and they are installed, activated and started through the GUI. Once started, the services expose their interfaces as Web Services through the Axis bundle. All modules in the middleware are multithreaded and use network sockets for interprocess communication, thus providing independence from development languages. The prototype interfaces the JXTA 2.4.1 overlay network.

To simplify the implementation, the DA sends simple URL-based WSDL requests, e.g. a query for the WSDL of a service called BookSearch has the form:
http://127.0.0.1:25766/axis/services/Book?wsdl.
Also, the current Composite Service Generator does not generate the Composite Service Bundle so we had to develop examples manually. The rest of the functionality described in Section 4 was fully implemented in the prototype.

To estimate the performance of our system, we first conducted experiments to compare the discovery and execution phases of p2pSOA-OJ with direct HTTP calls to the remote Axis engine. 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

Table 1: Comparison of p2pSOA with direct HTTP.

P2PSOA OPERATION	MEAN RTT (MSEC)	STD DEV
Discovery Phase	**172**	**31.78**
p2pSOA client-side	87	
p2pSOA server-side	64	
Axis WSDL response	13	
Communications	8	
Execution Phase	**76**	**15.95**
p2pSOA client-side	17	
p2pSOA server-side	42	
Axis SOAP response	11	
Communications	6	
HTTP OPERATION	**MEAN RTT (MSEC)**	**STD DEV**
HTTP/GET of WSDL (discovery)	108	20.27
HTTP/POST SOAP (execution)	35	16.89

as the client-side peer, as shown in the Figure 9(a). 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 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 to the server-side P2PA editing the WSDL file and to the client-side P2PA updating the MRT (see Figure 5). 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 CSB bundle to a remote service with our middleware and with direct HTTP/POST. Table 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.

Next, in order to assess the performance of our system as experienced by end-users, we conducted experiments using an example CSB application that we created called BookSearch. The scenario that we implemented is the following: Three friends decide to create a collaboration group to share access to their books. They run on their PCs a service that keeps a database of books available at their home. Friend A wants to look for a book, so she launches BookSearch and enters the desired book title. The CSB 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 CSB contacts the public Amazon.com Product Web Service to get pricing information about the book. The network setup where we implemented this scenario and conducted measurements is shown in Figure 9(b).

Table 2: Metrics for the BookSearch application.

BOOK SEARCH APPLICATION	MEAN TIME (MSEC)
Discovery Phase	**854**
WSDL retrieval RTT from Service B	250
WSDL retrieval RTT from Service C	275
WSDL retrieval RTT from Amazon	329
Stub Generation	**351**
Execution Phase	**470**
SOAP call RTT to Service B	125
SOAP call RTT to Service C	111
SOAP call RTT to Amazon.com	234

In the BookSearch application experiment after initialization, the CSB 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 2. The WSDL retrieval times and the SOAP calls for Services B and C are slower than the ones reported in Table 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.

We should mention that we experienced a long initialization phase delay (~7sec) in the above experiment. After dissecting this delay we found that it can be mostly attributed (~6sec) to the implementation of peer start-up in JXTA 2.4.1. This issue is known and has been addressed in the recently released JXTA version 2.5 [9]. Part of our immediate plans is to migrate to this newer release.

Finally, one of the major challenges in deploying software applications on mobile devices is to minimize the amount of memory (RAM) they consume. In order to get a sense of the suitability of our middleware design for such devices we measured the executable sizes and the memory footprints of the p2pSOA-OJ 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. We have not yet completed a comprehensive memory footprint analysis of the Axis engine (Web Services Adapter in the case of p2pSOA-OJ). Our findings are summarized in Table 3. The overall RAM size of our middleware is below 2 MB. Pending the results of the Axis analysis and given the trends in the mobile devices industry, we are optimistic that our design can be deployed on memory-constrained devices, such as mobile phones.

Table 3: Memory footprints of the p2pSOA.

SOFTWARE MODULE	EXEC SIZE (KB)	RAM (KB)
End-User Configuration Panel	7.1	61.1
Discovery Agent (DA)	42.0	284.3
Execution Agent (EA)	49.0	189.5
Runtime Adapter (RA)	29.6	76.3
WSDL2Java	6.9	560.6
P2P Adapter (P2PA)	149.0	240.5
p2pSOA-OJ Suite Totals	**283.6**	**1412.3**

6. CONCLUSIONS

We have designed a middleware architecture that enables the execution of composite services by combining private and public services over P2P overlay networks. Our middleware takes the P2P paradigm all the way up to the service level in the following manner: It routes SOAP messages end-to-end allowing the execution of composite services in a true peer-to-peer fashion; it does not require service-level intermediaries or centralized service registries; it utilizes off-the-shelf P2P technologies to deal with issues of pervasive service connectivity and leverage their capabilities in distributed group management and trust; it separates the runtime and connectivity layers, thus enabling us to accommodate different runtimes and P2P overlays. We have realized and tested an instance of the general architecture and a proof-of-concept prototype, which uses OSGi as the service container and runtime, and JXTA as the P2P overlay network without modifying them in any way.

Our immediate plans include the implementation of a WSA for exposing UPnP personal devices as Web Services, better integration of security (beyond those offered by JXTA and OSGi) at the *p2pSOA* middleware layer, and the definition of discovery and execution protocols that exploit the scalability of the underlying P2P overlay with respect to the number of devices and services in the trusted group. Other extensions under investigation include adding semantic discovery capabilities and integration of an open-source workflow editor or an end-user programming tool for capturing user intent.

REFERENCES

[1] Antoniu G, Cudennec, L., Jan, M and Duigou, M., "Performance scalability of the JXTA P2P framework", Proceedings IEEE Int'l. Symp. Parallel and Distributed Processing , Issue 26-30, pp. 1 – 10, 2007.

[2] Benatallah B., Sheng Q.Z. and Dumas M.,"The Self-Serv Environment for Web Services Composition", *IEEE Internet Computing*, 7(1):40- 48, 2003.

[3] Bradley, W.B. and Maher, D.P., "The NEMO P2P service orchestration framework", Proceedings 37th Annual Hawaii Int'l. Conf. on System Sciences, pp.10, 2004.

[4] Chen F-Y. and Yuan S-T., "A Contextualized Fault-tolerant Infrastructure for P2P Mobile", Proceedings IEEE Int'l. Conference on Services Computing, 2004, pp. 217-224.

[5] Conrad M., Dinger J., Hartenstein H., Schöller M. and Zitterbart M., "Combining Service-Orientation and Peer-to-Peer Networks", Proceedings of Kommunikation in VerteiltenSystemen (KiVS), pp.181-184, March 2005.

[6] Edrel M. et. al, "Web Service Gateway", Patterns: Service-Oriented Architectures and Web Services, IBM Redbooks, http://ibm.com/redbooks, 2004.

[7] Ferscha A. et. al., "A Light-Weight Component Model for Peer-to-Peer Applications", Proceedings 24th Int'l. Conf. on Distributed Computer Systems Workshops, pp. 520-527, 2004.

[8] Ford B .et. al., "Persistent Personal Names for Globally Connected Mobile Devices", Proceedings 7th USENIX Symposium on Operating System Design and Implementation, pp. 233-248, 2006.

[9] JXTA Community Projects, https://jxta.dev.java.net/, 2007.

[10] Jerstad I. et. al., "A service oriented architecture framework for collaborative services", 14th IEEE Int'l. Workshops on Enabling Technologies, pp.121–125, 2005.

[11] Kalofonos D. and N., Reynolds F., "Task-Driven End-User Programming of Smart Spaces Using Mobile Devices", Nokia Research Center, NRC-TR-2006-001, 2006.

[12] Knopflerfish OSGi, http://www.knopflerfish.org , 2007.

[13] Liu F. et. al., "TARGET: Two-way Web Service Router Gateway", on International Conference on Web Services, pp. 629-636, 2006.

[14] Loeser C. et. al, "Peer-to-Peer Networks for Virtual Home Environments", Proceedings 36th Hawaii Int'l.. Conference on System Sciences, vol. 9, pp. 282.3, 2003.

[15] Loureiro E. et. al., "A flexible middleware for service provision over heterogeneous pervasive networks", Int'l. Symposium World of Wireless, Mobile and Multimedia Networks, pp. 6, 2006.

[16] Manolakos E.S., Galatopoullos and D., Funk A., "Integrating Java and Matlab components into the same parallel and distributed application using *JavaPorts*. 18th Int'l. Parallel and Distributed Processing Symposium, pp.14b., 2004.

[17] Mondéjar R. et. al., "Enabling a Wide-Area Service Oriented Architecture through P2P Web Model", 15th IEEE Int'l. Workshops on Enabling Technologies, pp. 89-94, 2006.

[18] OASIS UDDI, http://www.uddi.org/, 2006.

[19] OSGi Alliance: OSGi Alliance. Official Web Site, http://www.osgi.org , 2007.

[20] Papazoglou M.P. and Heuvel W.J., "Service oriented architectures: approaches, technologies and research issues", *The VLDB Journal,* 16(3):389 – 415, 2007.

[21] SOAP Specifications, http://www.w3.org/TR/soap/ , 2004.

[22] Slominski A. et. al., "Asynchronous Peer-to-Peer Web Services and Firewalls", Proceedings of the 7th International Workshop on Java for Parallel and Distributed Programming, pp. 183.1, 2005.

[23] Vallée M., Ramparany F. and Vercouter L., "Flexible Composition of Smart Device Services", In Proc. of the 4th Int'l. Conf. on Pervasive Computing, pp. 91–96, 2006.

[24] Web Services Addressing (WSA), http://www.w3.org/Submission/ws-addressing/, 2007.

[25] Web Services–Axis, http://ws.apache.org/axis/, 2007.