Itinerary-Based Mobile Agent as a Basis for Distributed OSGi Services

Jonathan Lee, Shin-Jie Lee, Hsi-Min Chen, and Kuo-Hsun Hsu

Abstract—Recently, as more and more mobile devices embedded with intelligent software become part of our daily lives, Open Services Gateway initiative (OSGi) has gained increasing attention in the development of services for such devices. However, services residing in OSGi platforms are devised specifically for the platforms' local bundles. Although several works have enhanced OSGi with various communication mechanisms that enable remote service invocations, two crucial issues pertaining to mobile computing remain to be addressed: One is limited resources of mobile devices, and the other is unstable links experienced with mobile devices. To address these two issues, we propose an itinerary-based mobile agent approach with four major features: To implement mobile agents as OSGi bundles to deliver distributed OSGi services with lower resource consumption; to extend WSDL and WS-BPEL to help developers design mobile agent itineraries; to provide an opportunistic service matchmaking mechanism allowing mobile agents to adapt to context changes at runtime; and to devise asynchronous communication mechanism to enable reliable mobile agent transmissions among mobile devices with dynamic IP addresses and intermittent connection to the Internet. Through experimental evaluations, we demonstrate that both network traffic and turnaround time of our approach are better than those of client-server ones.

Index Terms—Mobile agent itinerary, OSGi, mobile agent

1 Introduction

Open Services Gateway initiative (OSGi) [27], an open service-oriented framework, has gained increasing attention in the development of services for mobile information devices. OSGi was originally designed as a software middleware in a residential gateway to remotely control home appliances [6], [36], and dynamically download and install required services from appliance providers via the Internet. Moreover, because of the benefits of openness, modularization, extensibility, and dynamic configurability, OSGi technology has being adopted by other domains, such as Eclipse plug-in development [13], [30], mobile devices [7], and vehicles [17].

OSGi utilizes a service-oriented architecture (SOA) to enable dynamic service invocations, whereby OSGi bundles can look up services in a service registry and invoke the selected services at runtime. However, the current OSGi framework by itself does not support access to remote services because its focus was distributed systems, namely, services residing on an OSGi platform are developed specifically for the platform's local bundles. This limitation poses a critical issue for distributed systems that need to use services located on remote OSGi platforms. To address this issue, several works [29], [14], [16], [15], [38], [8] have extended OSGi with various communication mechanisms to enable remote service invocations.

The problems with these extensions can be best described as follows: If an OSGi application executed on a mobile device requires multiple services from remote OSGi platforms, two crucial issues arise:

- *Limited resources*. Mobile devices have limited resources, such as computing power, memory, and bandwidth for delivering services. If developers use the traditional client-server paradigm to access remote OSGi services, the network's consumption becomes heavier as the number of accessed remote services increases. Consequently, relying on a mobile device to coordinate the delivery of multiple distributed service invocations creates a performance bottleneck.
- *Unstable link*. Mobile devices are usually supported by constantly changing network connections. Thus, creating a network connection for each remote service accessed by a mobile device increases the possibility of service execution failure when a link becomes unstable. If one of the connections between a mobile device and a remote OSGi platform disconnects unexpectedly, applications using the service may need to be reexecuted from scratch.

To address the problems, we propose an itinerary-based mobile agent approach, called MA-OSGi, which facilitates OSGi service delivery on mobile devices. Four key features of the approach are described as follows:

First, we implement a mobile agent as an OSGi bundle to access distributed OSGi services. As shown in Fig. 1, we have developed a mobile agent framework on top of OSGi.

- *J. Lee is with the Department of Computer Science and Information Engineering, National Taiwan University, Taipei 10617, Taiwan. E-mail: jlee@csie.ntu.edu.tw.*
- *S.-J. Lee is with the Computer and Network Center, National Cheng Kung University, Tainan 701, Taiwan. E-mail: jielee@mail.ncku.edu.tw.*
- *H.-M. Chen is with the Department of Computer Science and Information Engineering, National Central University, Jhongli 32001, Taiwan. E-mail: seeme@selab.csie.ncu.edu.tw.*
- *K.-H. Hsu is with the Department of Computer Science, National Taichung University, Taichung 40306, Taiwan. E-mail: glenn@mail.ntcu.edu.tw.*

Manuscript received 25 Oct. 2011; revised 21 Mar. 2012; accepted 12 May 2012; published online 23 May 2012.
Recommended for acceptance by D. Talia.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-10-0761.
Digital Object Identifier no. 10.1109/TC.2012.107.

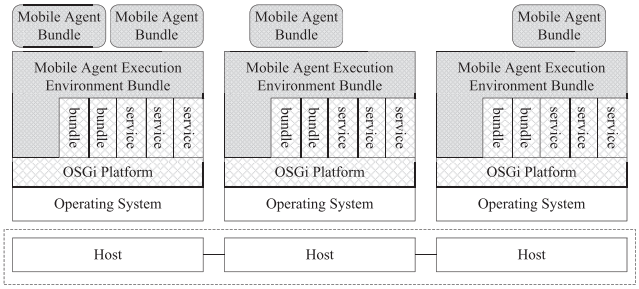

Fig. 1. The relationship among OSGi framework, MAEE, and mobile agent.

In each OSGi platform, we implement a mobile agent context as an OSGi bundle called a Mobile Agent Execution Environment (MAEE) bundle. MAEE facilitates mobile agent migrations between OSGi platforms and manages the life cycle of mobile agents working on it. Additionally, a mobile agent is designed as an OSGi bundle. A mobile agent can travel to the distributed OSGi platforms specified in its itinerary, access services and resources residing on those platforms, execute the assigned tasks, and then bring the results back to the user.

Second, mobile devices may appear and disappear unexpectedly due to unstable links and changes in IP addresses. Thus, based on the proposed asynchronous communication mechanism, we implement an OSGi communication service, called a docking service, in each OSGi platform to ensure reliable mobile agent migration between OSGi-enabled mobile devices and the target platforms.

Third, our approach helps developers concentrate on the design of mobile agent itineraries from the perspective of high-level processes, rather than writing mobile agent code. Although effort has been made in modeling location changes of mobile agents based on Petric nets [37], our work focuses more on providing a mobile agent architecture based on two *de facto* standards, OSGi and WS-BPEL. Moreover, developers only need to design mobile agent itineraries through a graphical design tool [23].

Finally, we leverage our previous work, PPN service matchmaker [24], [26], to help mobile agents adjust to the context changes dynamically without redesigning mobile agent itineraries.

We evaluate the performance of MA-OSGi by comparing its network traffic and turnaround time with two client-server approaches: R-OSGi [29] and SOAP-OSGi [16]. The evaluation result shows that MA-OSGi not only generates less network traffic between a mobile device and servers in a mobile computing environment, but also exhibits a shorter turnaround time in the life cycle of service invocations than R-OSGi and SOAP-OSGi.

The remainder of the paper is organized as follows: Section 2 contains a review of approaches for the delivery of distributed OSGi services. The life cycle of an itinerary-based mobile agent is introduced in Section 3. Section 4 elaborates on the proposed itinerary-based mobile agent architecture. In Section 5, we conduct a series of experiments to compare our approach with other methods with respect to four evaluation metrics. Finally, in Section 6, we conclude with the benefits of our proposed approach and our future research plan.

2 Related Work

Several approaches have been proposed for the delivery of distributed OSGi services. For example, R-OSGi [29], a distributed middleware built on top of OSGi platforms, facilitates access to distributed OSGi services. However, as service proxies are generated by clients, the workload on clients will increase as the demand for remote services increases.

Eclipse Communication Framework (ECF) [32] allows multiplex transport implementations, such as R-OSGi, ECF Generic, JavaGroups, Java Message Service (JMS), Extensible Messaging and Presence Protocol (XMPP), and Skype, to access remote services. Nevertheless, the problem of ECF is that it is tightly bound to Eclipse so that other OSGi implementations, for example, Apache Felix and Knopflerfish, are failed to directly adopt it to invoke remote OSGi services.

Ibramin and Zhao [14] use a *D-OSGi* bundle to realize the following code mobility paradigms defined in [11]. Like R-OSGi, introducing remote evaluation and code on demand into the interactions between clients and servers also increases the resource consumption of mobile devices.

Knopflerfish [16] includes an Axis bundle that can transform OSGi services into web services. However, if a large amount of data, such as multimedia or scientific data, is transferred between OSGi platforms in Simple Object Access Protocol (SOAP), the performance will be degraded due to the XML encoding overhead [5].

Communication Service Concierge [38] and Extended Service Registry [15] are solutions that use Java RMI technology to facilitate access to remote OSGi services. However, to transform OSGi services into RMI services, developers have to modify and recompile code of existing OSGi services.

P2Pcomp [8] leverages OSGi as the component management framework; and P2P implementations, for example, JXTA, serve as communication channels to interact with remote components and construct mobile P2P applications. To invoke remote services, P2Pcomp adopts Java Dynamic Proxy to generate client-side proxies on demand.

Wu et al. [36] propose a service-oriented, smart-home architecture based on OSGi and mobile agent technology. The primary differences between Wu et al. work and ours are listed as follows: 1) for mobile agent implementation, Wu's approach employs MASML to implement text-format mobile agents whereas our approach leverages OSGi bundles to realize binary-format ones; 2) for mobile agent execution, Wu's approach requires Agent Specification Interpretation Algorithm (ASIA) to interpret MASML mobile agents whereas our approach directly adopts the JVM to run mobile agents; and 3) to process data or react to the results of service executions, Wu's approach utilizes European Computer Manufacturers Association script whereas our approach uses standard BPEL language.

Table 1 summarizes the technical features of the distributed OSGi technologies. The comparison focuses on the following aspects:

- *Service description*. To support the development of large-scale distributed systems, an expressive

TABLE 1
Distributed OSGi Technologies

	service description	global service registry	service invocation	service invocation flow	asynchronous invocation	adaptive to context change
R-OSGi [29]	qualified interface name + service properties	SLP	proxy of remote service	n/a	asynchronous service invocation	n/a
ECF [32]	qualified interface name + service properties	zeroconf/SLP/ Apache Zookeepe	proxy of remote service	n/a	asynchronous service invocation	n/a
D-OSGi [14]	service name	n/a	proxy of remote service + service/bundle migration and cloning	n/a	publish-subscribe	n/a
Knopflerfish [16]	WSDL	n/a	SOAP proxy of remote service	n/a	n/a	n/a
Communication Service Concierge [38]	service name	RMI registry	RMI proxy of remote service	n/a	n/a	n/a
Extended Service Registry [15]	service name	Extended RMI registry	RMI proxy of remote service	n/a	n/a	n/a
P2Pcomp [8]	service name	n/a	dynamic generated proxy of remote service	n/a	asnycInvoke method	n/a
Wu *et al.* [36]	service name	Agent Directory	mobile agents	MASML	n/a	n/a
Itinerary-Based Mobile Agent	WSDL	UDDI	mobile agents	BPEL-based itinerary	docking service	PPN service matchmaker

language that describes services is needed to bind remote services. Most approaches assume that developers know exactly what the remote service they want and which methods can be invoked, and therefore, they use the original service naming mechanism provided by OSGi. However, simply specifying service names is insufficient to select and distinguish between the services offered by the huge number of distributed providers [31].

- *Global registry*. A global registry provides the registration and discovery support that bridges service requesters and providers in distributed systems. Most of the approaches listed in Table 1 only register service names, or service properties, in global registries. They do not consider the methods and parameters of the services. In addition, some of the approaches do not address the issue of service discovery. By contrast, our approach extends UDDI, which supports fine-grained service discovery, to register OSGi services so that users can obtain their requested services.
- *Service invocation*. Most of the works in Table 1 use client-side proxies to invoke remote services; however, managing proxy life cycles is problematic for service clients. As the number of requested remote services increases, clients need more memory/storage to keep the corresponding service proxies. By contrast, in our approach, the implementation of mobile agents for various service invocations is always the same with each other. The key difference between our approach and others is the embedded itineraries. Therefore, a client only need to keep a JAR file of a mobile agent with a footprint around 9 KB and to maintain XML-based itineraries for service invocations. After sending out the mobile agent, the memory occupied for the purpose of mobile agent instantiation in the client is released. Furthermore, managing high-level invocation processes represented by mobile agent itineraries is much easier for clients than managing low-level service proxies.
- *Service invocation flow*. Service invocation flow specifies structures, such as the sequence, branch, loop, cases, and parallel structures, to invoke multiple services. Except for our approach and that of Wu et al., the approaches in Table 1 focus on creating a connection to invoke a remote service. These approaches do not consider how to compose multiple distributed services. Thus, if an application includes more than one remote service, the application developer writes the service flows in client-side code and adopts the client-server paradigm to access remote services. As mentioned earlier, however, the client-server paradigm is less suitable for mobile computing environments than the mobile agent paradigm.
- *Asynchronous invocation*. Asynchronous invocation allows clients not to block during remote invocations. By means of asynchronous support, loose coupling between clients and remote services can be achieved. In Table 1, only our approach, R-OSGi, D-OSGi, and P2Pcomp support asynchronous invocation. In our approach, although services are invoked synchronously by mobile agents at target OSGi platforms, from the viewpoint of clients, remote services are invoked in an asynchronous manner where clients can process the main program flows intermediately without waiting for the returns of mobile agents after the mobile agents are launched to invoke remote services.
- *Adaptive to context change*. If mobile agents are adaptive to context changes, services can be invoked in a more flexible way, that is, service selection decisions can be made at runtime based on the current context. In Table 1, only our approach takes this feature into account. Since contexts change constantly in mobile computing environments, rewriting programs for each context change is not applicable for application developers.

3 Life Cycle of the Itinerary-Based Mobile Agent

We begin this section by explaining why OSGi platform is a promising candidate to serve as a basis for mobile agent's context. Then, the main problem of and our solution to

adopting OSGi platform as mobile agent's context to perform tasks collaboratively in distributed devices are discussed in details.

3.1 OSGi Framework as a Basis for Mobile Agent's Context

In recent years, several mobile agent technologies and systems have been proposed, but the applications of mobile agents are scarce. The major obstacle to the widespread use of mobile agents is that there is no open framework as a basis for mobile agent's context on which agents can interoperate in a collaborative fashion. There are a variety of context bases adopted in various mobile systems; for example, AWK Interpreter is used in Transportable Agent [19], TCL Interpreter is employed in Agent TCL [12], Java Virtual Machine (JVM) is exploited in Ajanta [1], Grasshopper [3], Aglets [21], and JIMAF [9] that is a software layering approach to the integration of heterogeneous Java-based mobile agent systems, and C/C++ Interpreter is adopted in Mobile-C [4].

In these existing approaches, interactions among mobile agents, services, and execution environments are specified in code-level languages rather than a high-level description language, such as WSDL or BPEL. In our approach, services and execution environments are specified by means of an extension to WSDL, and interactions among the mobile agents, services, and the environments are described as a mobile agent's itinerary through an extension to WS-BPEL. Therefore, mobile agent developers are empowered to specify the interactions among these components in a systematic manner from a high-level perspective.

OSGi offers the following two valuable key features that make it a promising candidate to serve as a context for mobile agents:

1. *Modularized programming paradigm*. Although JVM is the most adopted context basis in many mobile agent systems, it has two crucial limitations [21]. The first problem is "*no protection of references*," that is, a Java object's public methods are available to any other object that has a reference to it; and therefore, there is no way to exercise access control between a mobile agent and other objects. OSGi provides a modularized programming paradigm that enables the encapsulation of mobile agents, execution environments, and services as bundles, and the specification of import/export statements in bundles to control access between them.
2. *Life-cycle management*. The second problem is "*no object ownership of references*," that is, no one owns the references to a given object in Java. This reveals a loophole that an agent/object has been destroyed may be turned to keeping alive by other object that has a reference to it. OSGi provides a rigid life-cycle management for bundles (see Fig. 2a), in which once a bundle is uninstalled, others are not allowed to have any access to it.

3.2 OSGi and Mobile Agents

Fig. 2 shows the life cycle of an itinerary-based mobile agent, in which Fig. 2a depicts the life cycle of a general OSGi bundle, and Fig. 2b shows the additional states specific to a mobile agent bundle. When an execution environment of mobile agents receives a mobile agent bundle from a remote OSGi platform, it installs the bundle on the local OSGi platform and checks that the bundle's installation constraints are satisfied. If the result is positive, the bundle is activated to deserialize the state of the mobile agent and then performs the specified task on the local OSGi platform. Following completion of the task, the mobile agent state with the relevant data is serialized into the mobile agent bundle, and the bundle is uninstalled from the platform. The execution environment of mobile agents then transfers the bundle to the next OSGi platform.

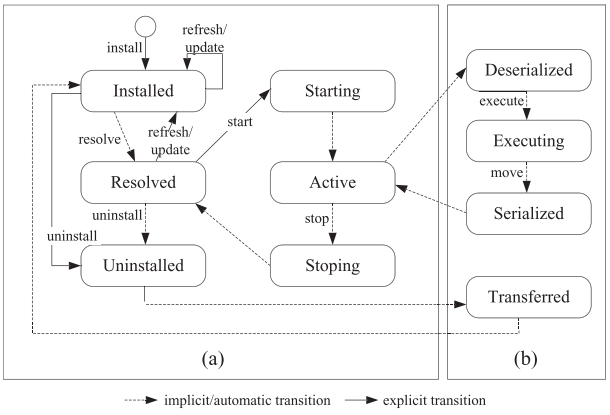

Fig. 2. Life cycle of the Itinerary-based mobile agent.

4 Itinerary-Based Mobile Agent Architecture

In this section, we detail the implementations of each component of the itinerary-based mobile agent architecture. First, we elaborate the internal structure of the mobile agent execution environment and the mobile agent that are implemented as OSGi bundles. Second, we explain how to apply standard description languages to describe OSGi services and a mobile agent itinerary. Finally, the opportunistic service matchmaking mechanism for agents to determine the traversed paths in a dynamic manner, and the asynchronous communication mechanism to enable reliable mobile agent transmission are discussed.

4.1 MAEE Bundle

Mobile agent systems are required to provide a basic host environment to enable an agent to perform tasks on the visited sites. This environment is known by various names, such as a *place* in [2], an *agent server* in [12], [35], [18], [1], a *core* in [28], or an *agent context* in [22]. In our work, it is called an MAEE.

Fig. 3 shows the architecture of our itinerary-based mobile agent system, where an MAEE bundle is installed on top of each OSGi platform to facilitate the communication among OSGi platforms and manage the life cycle of mobile agents that perform tasks on the MAEE. Detail descriptions of each component are explained below.

MAEE Activator is an entry point to initiate the execution of a mobile agent environment bundle. In OSGi, each bundle has to write an activator class to start and stop the inner workings of the bundle. A start method in the MAEE activator is implemented to instantiate the services

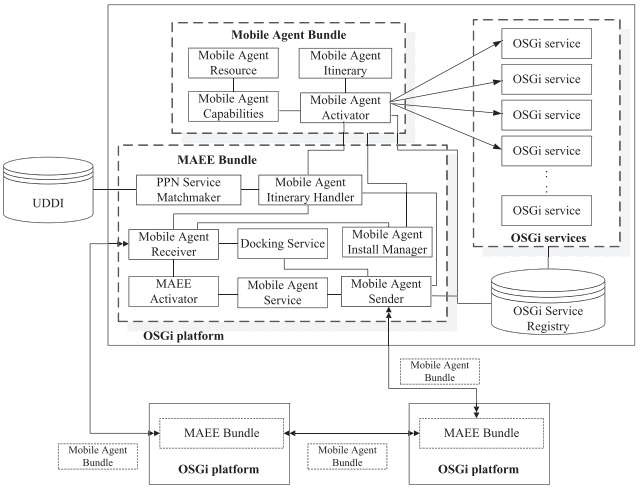

Fig. 3. Itinerary-based mobile agent architecture.

required for processing mobile agents; meanwhile, a stop method to reclaim the resources from these services while the MAEE is terminated.

Mobile Agent Receiver is a daemon service that listens to the arrival of mobile agents sent by remote OSGi platforms. When this component receives a mobile agent, it will check the agent's itinerary to determine which task to perform on the target OSGi platform.

Mobile Agent Service is an OSGi service that provides an interface for external bundles to launch mobile agents and notify events when the agents return. Through the interface, developers can implement OSGi applications that leverage mobile agents to deliver services provided by remote OSGi platforms.

Possibilistic Petri-Net (PPN) Service Matchmaker is an OSGi service that leverages PPN formalism to facilitate mobile agents to determine the next visited platforms at runtime. When a mobile agent leaves a platform and finds that the next platform is not specified in its itinerary explicitly, it informs PPN Service Matchmaker to select an appropriate platform whose service meets the constraints specified in the matchmaking request. The detail of PPN Service Matchmaker will be presented in Section 4.5.

Docking Service is an OSGi component that supports asynchronous communication among OSGi-enabled devices. The service enables reliable mobile agent migration across OSGi platforms even if some network links are unstable. An OSGi platform deployed on a stationary machine with a stable network connection can act as a docking server on behalf of a set of mobile devices. The docking service maintains a registry of the profiles of known mobile devices, and temporarily stores received mobile agents that intend to migrate to those devices in the case that the connections to the devices are currently unavailable. After the devices reconnect to the docking server, the docking service retrieves mobile agents from the queues and sends them to the target mobile devices.

Mobile Agent Install Manager deals with the installation of mobile agent bundles on an OSGi platform. When a mobile agent receiver receives a mobile agent bundle from a remote OSGi platform, it will check whether the visited device has already been specified in the mobile agent's itinerary or not. If the response is positive, the receiver forwards the bundle to the mobile agent install manager to install the mobile agent on this platform. The installation process involves two steps: 1) The first step is to modify the manifest of the mobile agent bundle to include the package name of the service invoked on the target platform. This is due to the fact that the OSGi specification requires that a bundle specifies the package names of services to be invoked in its manifest. 2) The second step is to install the mobile agent bundle.

Mobile Agent Itinerary Handler is a key component in an MAEE bundle to interpret the itinerary of mobile agents to know what to perform on the current platform and where to go next.

Mobile Agent Sender transfers mobile agents to the next site specified in their itineraries. After a mobile agent completes its task on the current OSGi platform, the mobile agent sender is requested to transfer the mobile agent to the next site. If the connection to the next site cannot be established immediately, the mobile agent sender redirects the mobile agent to the docking service, which will then handle the transmission in due course.

4.2 Mobile Agent Bundle

In this section, the implementation of the itinerary-based mobile agent through the use of mobile agent bundle is explained in detail. See Fig. 3 for the components of a mobile agent bundle.

Mobile Agent Activator is an entry point to initiate the execution of a mobile agent bundle. After an MAEE's mobile agent install manager installs received mobile agent bundle, the start method of the mobile agent bundle is automatically invoked to activate a mobile agent. The start method first recovers the mobile agent's state from the bundle and then informs the MAEE's mobile agent itinerary handler to obtain the service references—the mobile agent wants to invoke on this site, from the agent's itinerary. After invoking the services, the mobile agent serializes the return values and its state into the bundle and requests the MAEE's mobile agent sender to uninstall the mobile agent bundle and transfer it to the next site.

Mobile Agent Capabilities are a set of operations, including their implementations, defined in the Java class that implements BundleActivator in a mobile agent's bundle. The capabilities can be specified in an agent's itinerary and then be invoked at runtime. For example, if a mobile agent's bundle contains a class agent - CarAccidNotificationAgent, including a public operation resizeImage for resizing an image, the operation can be specified as follows in the agent's itinerary and then be invoked at runtime: <invoke operation="resizeImage" outputVariable="ImageBean" inputVariable="ImageBean" partnerLink="agent_capability" portType="agent.CarAccidNotificationAgent" />

To reduce the workloads of mobile devices or utilize the resources of remote OSGi platforms, mobile agent developers can implement a number of functions as mobile agent capabilities resided in a mobile agent's bundle. The capabilities are specified in the agent's itinerary to indicate which capability will be utilized when visiting a site.

Mobile Agent Resources are the resources needed by mobile agent capabilities to perform a certain task on an

OSGi platform. The resources can be any artifact, such as images and relevant data, archived in a mobile agent bundle.

Mobile Agent Itinerary is a specification that describes the path traversed, services accessed, and data carried by a mobile agent. Instead of hard coding an itinerary in a mobile agent, in this work, we develop an XML-based itinerary construct that enables mobile agent developers to design a mobile agent's itinerary in a flexible manner, which will be discussed in the Section 4.4.

4.3 Describing OSGi Services with WSDL

Our proposed mobile agent's itinerary guides the agent's movements and whereabouts of the services that are needed for the agent to access. Before composing a mobile agent itinerary, developers need to determine the locations of the required services. However, there are two constraints with the current OSGi specification that impedes the access to multiple distributed OSGi services:

- It only supports a stand-alone registry to provide registration, deregistration, and discovery functions on an OSGi platform. Consequently, bundles outside an OSGi platform would not be able to discover the services offered by the platform.
- The OSGi registry only records the qualified class name and user-specified properties for each registered service. As a result, crucial information about the internal structure of a service, such as the signatures of methods and the types of parameters, is not provided.

To address the problem, we extend OSGi to enrich description and discovery of OSGi services by introducing Web Service Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI). WSDL is a standard XML-based language usually used to describe the access interface of web services, including the definitions of services, operations, and messages. It also allows users to specify the binding mechanism, including the description of the binding style and transport protocol used to access services. The benefits of adopting WSDL in this work are: 1) it is independent of any programming language; 2) it has been used extensively to describe services; and 3) it has more expressive power than the OSGi registry.

In this work, we adopt UDDI affiliated service registries [33] in which service providers can publish their OSGi service descriptions as WSDL documents and service requesters can search for these services. A service requester can be either a mobile agent developer that uses the UDDI Browser provided by the Itinerary Designer to search for required services (refer to Section 4.4) or a PPN Service Matchmaker that allows mobile agents to adjust to context changes dynamically without redesigning mobile agents' itineraries (refer to Section 4.5). In UDDI 3.0 [33], a registry affiliation mechanism is proposed to provide service discovery on a more broadly distributed environment, by which a publisher may have the authority to add many entities to a registry and, as such, a publisher could potentially publish the entirety of a registry's contents into another registry by effectively mirroring the data.

```
1 <element name="binding" type="bindingType"/>
2 <complexType name="bindingType">
3   <attribute name="style" type="styleChoice"/>
4   <attribute name="transport" type="uriReference"/>
5 </complexType>
6 <simpleType name="styleChoice">
7   <restriction base="string">
8     <enumeration value="rpc"/>
9     <enumeration value="document"/>
10    <enumeration value="osgi_rpc"/>
11  </restriction>
12 </simpleType>
```

Fig. 4. XML schema for WSDL extension.

Additionally, data can be shared between two UDDI registries by registry-to-registry data sharing.

To adopt WSDL for OSGi services, the binding part of WSDL needs to be altered. In WSDL, the binding description of a web service includes the binding style (e.g., rpc or document), the transport protocol (e.g., HTTP or SMTP), and the location of the service. The alterations include the following: 1) the binding style is changed to `osgi_rpc` (see Fig. 4), 2) the transport protocol is set as `matp` to transfer itinerary-based mobile agents between OSGi platforms, and 3) the location description for an OSGi service is modified with the prefix `matp` to mark the service accessed by an itinerary-based mobile agent.

Refer to the first alternation (Fig. 4), an extension to WSDL is made by defining an XML schema for the binding style of OSGi services. A new kind of binding style—`osgi_rpc`—is included in the style attribute in addition to `rpc` and `document`. The URI-scheme name of the transport protocol and location description is set as `matp` for the second and third alternation.

4.4 Describing Mobile Agent Itinerary with WS-BPEL

A mobile agent itinerary is used to describe path traversed, services accessed, and data carried by a mobile agent. Most existing mobile agent systems hardcode their itineraries with their mobile agents' source code, which makes the update of an itinerary seem to be almost impossible without going through the recompilation of their entire source code.

In this work, our objective is to facilitate mobile agent developers to design a mobile agent itinerary by specifying its high-level processes rather than by means of programming. To separate the itinerary from the source code of mobile agent, an extension to Web Service Business Process Execution Language (WS-BPEL) is made to parse an itinerary in the MAEE.

Fig. 5 shows constructs of a mobile agent itinerary. There are three main parts in the original WS-BPEL specification: 1) the partner links, specifying the involved services; 2) the variables, designating the data types used for service invocations; and 3) the business logic, depicting the flow structure of service invocations. Our extension to the specification includes two additional elements: A `profile` element to indicate a mobile agent's unique identifier, and a `pointer` element to show which OSGi platform a mobile agent is currently working on. Fig. 6 shows the formal definitions of the `profile` and `pointer` elements defined in XML Schema.

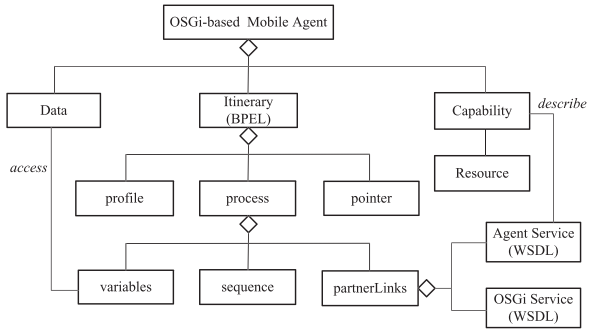

Fig. 5. Constructs of the mobile agent itinerary.

Instead of composing a mobile agent's itinerary manually, we have developed a tool called *Itinerary Designer* to help developers design itineraries. To compose an itinerary, a developer can use the *UDDI Browser* provided by the *Itinerary Designer* to search for required services and import their descriptions from UDDI servers. The developer can then drag and drop the imported services into a workspace and edit the control flows of the service invocations. Finally, when developers finish designing mobile agents' itineraries, the *Itinerary Designer* automatically generates mobile agent bundles with the designed itineraries.

4.5 Opportunistic Service Matchmaking Mechanism

In the previous scenario, we assumed that mobile agents migrate across OSGi platforms and access remote services based on their itineraries compiled at design time. However, specifying fixed services in the itineraries may reduce the mobile agents' flexibility because not all services are determined at the design time till mobile agents work on their assigned tasks. Predetermined itineraries mean that mobile agents cannot adapt to context changes and fail to select appropriate services based on intermediate results obtained while they are migrating. In general, redesigning mobile agent itineraries for each context change is not only clumsy but also unacceptable for itinerary design.

To address the problem, we provide an opportunistic service matchmaking mechanism to allow mobile agents to adjust to context changes dynamically without redesigning mobile agents' itineraries. An OSGi service may have several implementations with various context-aware properties, such as location, valid period, and security, provided by different OSGi platforms. Therefore, if we can select an appropriate service that meets the guard condition of the current context, the next OSGi platform that provides the selected service can be determined at runtime for a mobile agent.

We extend the WS-BPEL specification to allow an itinerary designer to include matchmaking requests in a mobile agent's itinerary. Itinerary designer can make a runtime matchmaking request in an itinerary in which the visiting platform is not specified in advance, namely, the value of the attribute `partnerLink` is left empty. We add a new attribute `constraints` to the `invoke` element. Itinerary designer can specify a set of the pairs of property and value in the `constraints` attribute for runtime service matchmaking.

In addition to the extended WS-BPEL specification, we adopt our previous work, *PPN Service Matchmaker* [26], to support service matchmaking at runtime. PPN [24] is an

```
1 <element name="profile" type="profileType"/>
2 <complexType name="profileType">
3   <attribute name="homeName" type="string"/>
4   <attribute name="id" type="string"/>
5 </complexType>
6
7 <element name="pointer" type="pointerType"/>
8 <complexType name="pointerType">
9   <attribute name="invoke" type="string"/>
10 </complexType>
```

Fig. 6. XML schema that defines the `profile` and `pointer` elements used to describe mobile agent itineraries.

extension of Petri-nets with the capability to add possibility Π and necessity N measures on the possibilistic tokens and possibilistic transitions. It uses possibility distributions over all places and tokens to display the uncertainty about possible locations of a token before receiving certain information. The calculation of (N, Π) of corresponding possibilistic tokens and possibilistic transitions is based on the possibilistic reasoning. Fig. 4 is a representation of matching request with service in PPN-ASDL. The general description of the matchmaking algorithm is to

1. construct the PPN for the request and the service,
2. create new possibilistic transitions between corresponding places in these two PPNs,
3. put a possibilistic token in each input places of request,
4. fill in the initial (N, Π) of each possibilistic tokens and possibilistic transitions, and
5. fire the transitions and calculate the (N, Π) of the remaining possibilistic tokens and possibilistic transitions according to the possibilistic reasoning rule.

The (N, Π) of the possibilistic token in the output place of the request will be the confidence degree that the service can satisfy the request. Please refer to [25] for more detail about this algorithm.

Thus, when leaving a platform, if a mobile agent finds that the next platform to be visited in its itinerary has yet to be determined, it can inform *PPN Service Matchmaker* to find an appropriate service based on the given condition. *PPN Service Matchmaker* would then return the address of the service that meets the condition. Finally, the mobile agent moves to that address.

4.6 Asynchronous Communication Mechanism

Usually, mobile devices used in mobile computing environments are not equipped with fixed IP addresses and cannot connect to the Internet continuously. A problem may occur when an agent wishes to return to its home device but encounters a situation, where the IP address of the device has been changed or disconnected from the Internet. To address the problem, an asynchronous communication mechanism (called docking service in MAEE) is devised to provide reliable communications between mobile devices and service platforms.

Fig. 7 shows the sequence diagram of the proposed asynchronous communication protocol. When a mobile device indicates its intention to join an itinerary-based mobile agent system, it registers its profile with a docking server. When the server accepts the request, it will try to forward the mobile agent to the mobile device. If the connection between the docking server and the mobile device is available, the

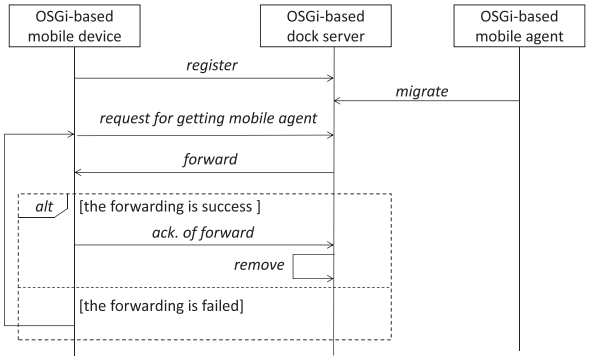

Fig. 7. The proposed asynchronous communication protocol.

docking client on the mobile device sends a request for getting a mobile agent to the docking server and then the server forwards the mobile agent to the mobile device directly. If the forwarding succeeds, that is, all bytes of the mobile agent bundle have been sent to the mobile device, the dock client then sends an acknowledge message to the server. Once the server receives the message, it will remove the mobile agent from its queue. If the forwarding fails, the client will try to resend the request for getting the mobile agent and receive the forwarded data repeatedly until the mobile agent is forwarded successfully.

5 Experimental Evaluation

In this section, we evaluate the performance of MA-OSGi by comparing its performance for distributed OSGi services with those of client-server approaches. We define two evaluation metrics: network traffic and turnaround time, as a basis for comparing MA-OSGi with R-OSGi and SOAP-OSGi.

5.1 Definition of Evaluation Metrics

Even though mobile agent approach gains several advantages in distributed computing environments, not all applications are appropriate to apply it for access to remote services. Generally, service invocation processes can be categorized into two types, embarrassingly parallel and workflow [10]. No dependency between parallel tasks/services exists in embarrassingly parallel process, while there exist dependencies among tasks/service in workflow process. For the applications of embarrassingly parallel process, because mobile agents have to carry results back to clients per remote service invocation, which incurs more network load than client-service approaches, mobile agents are not applicable in such applications as compared with client-service approaches. Therefore, we compare our approach with others within the scope of workflow-based applications in the following evaluations.

Because a workflow process, constituted by various flow structures, such as loop and branch, may be very complex, we assume that the constituted flow structures can be reduced to a high-level sequence flow to provide a general comparison. Thus, we use a sequential service invocation process as an illustration for evaluations. To evaluate the performance of the client-server and the mobile agent paradigms, two metrics are adopted: 1) network traffic: Data transmitted over a network [34]; and 2) turnaround time: The total time taken between the submission of a program for execution and the return of the complete output to the customer [20].

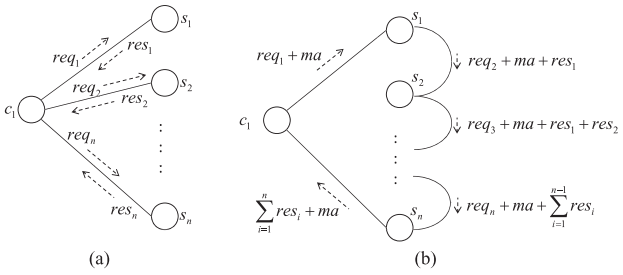

Fig. 8. The network traffic generated by (a) the client-server paradigm, and (b) the mobile agent paradigm.

The network traffic metric is formally defined in Definition 1.

Definition 1. *Given a set of OSGi servers $S = \{s_1, \ldots, s_n\}$ providing OSGi services requested by a client, the client-side network traffic in the client-server paradigm, denoted as N_c^{cs}, the overall network traffic in the client-server paradigm, denoted as N_{total}^{cs}, the client-side network traffic in the mobile agent paradigm, denoted as N_c^{ma}, and the overall network traffic in the mobile agent paradigm, denoted as N_{total}^{ma}, are defined as follows:*

$$N_c^{cs} = N_{total}^{cs} = \sum_{i=1}^{n} req_i + \sum_{i=1}^{n} res_i \quad (1)$$

$$N_c^{ma} = 2 \cdot ma + req_1 + \sum_{i=1}^{n} res_i \quad (2)$$

$$N_{total}^{ma} = N_c^{ma} + \sum_{i=2}^{n} \left(req_i + ma + \sum_{j=1}^{i-1} res_j \right), \quad (3)$$

where req_i is the size of the request message requesting a service provided by s_i, res_i is the size of the reply message returned by the server s_i, and ma is the size of the message that encapsulates the body of a mobile agent.

As shown in Fig. 8a, the client sends a request message to each remote service for a service invocation and receives a response from the invoked service in the client-server paradigm. The client-side network traffic, N_c^{cs}, counts the amount of message size sent and received by the client, which is equal to the overall network traffic, N_{total}^{cs}.

As for the mobile agent paradigm, as depicted in Fig. 8b, a mobile agent travels among servers with no need to return to the client for each service invocation; and therefore, the client-side network traffic, N_c^{ma}, considers only the outgoing mobile agent, i.e., $ma + req_1$, and the incoming mobile agent that carries the executed result back to the client, i.e., $ma + \sum_{i=1}^{n} res_i$. The overall network traffic in the mobile agent paradigm, N_{total}^{ma}, amounts to the network traffics that a mobile agent traverses through all the servers, i.e., $\sum_{i=2}^{n}(req_i + ma + \sum_{j=1}^{i-1} res_j)$, plus the network traffic of the client.

Turnaround time is a derived metric from transmission time and processing time. The transmission time—the amount of time spent for a message to travel from the start

node of a link to its destination node—is defined in Definition 2. Meanwhile, Definition 3 defines the processing time—the amount of time spent to process service invocations, tackle incoming messages, and obtained results.

Definition 2. *Given a set of OSGi servers $S = \{s_1, \ldots, s_n\}$ providing OSGi services requested by a client, the total transmission time spent in the client-server paradigm, denoted as T^{cs}, and the total transmission time spent in the mobile agent paradigm, denoted as T^{ma}, are defined as follows:*

$$T^{cs} = \sum_{i=1}^{n} \frac{req_i + res_i}{b_{c,s_i}} \tag{4}$$

$$T^{ma} = \frac{req_1 + ma}{b_{c,s_1}} + \frac{ma + \sum_{i=1}^{n} res_i}{b_{c,s_n}} + \sum_{i=1}^{n-1} \frac{req_{i+1} + ma + \sum_{j=1}^{i} res_j}{b_{s_i,s_{i+1}}}, \tag{5}$$

where b_{c,s_i} is the bandwidth between the client and server s_i and $b_{s_i,s_{i+1}}$ is the bandwidth between server s_i and server s_{i+1}.

With respect to the client-server paradigm, the transmission time, T^{cs}, is the amount of time spent in sending and receiving messages for all service invocations. In the case of the mobile agent paradigm, the transmission time, T^{ma}, consists of sending a mobile agent with related data to the first server, i.e., $\frac{req_1+ma}{b_{c,s_1}}$, receiving a mobile agent with related data from the last server, i.e., $\frac{ma+\sum_{i=1}^{n} res_i}{b_{c,s_n}}$, and transmitting a mobile agent with related data among servers, i.e.,

$$\sum_{i=1}^{n-1} \frac{req_{i+1} + ma + \sum_{j=1}^{i} res_j}{b_{s_i,s_{i+1}}}.$$

Definition 3. *Given a set of OSGi servers $S = \{s_1, \ldots, s_n\}$ providing OSGi services requested by a client, the processing time spent in the client server paradigm, denoted as P^{cs}, and the processing time spent in the mobile agent paradigm, denoted as P^{ma}, are defined as follows:*

$$P^{cs} = \frac{(\sum_{i=1}^{n} mp_i) + rp}{c_c} + \sum_{i=1}^{n} \frac{sp_i}{c_{s_i}} \tag{6}$$

$$P^{ma} = \frac{mp_n + rp}{c_c} + \sum_{i=1}^{n-1} \frac{mp_i}{c_{s_i}} + \sum_{i=1}^{n} \frac{sp_i}{c_{s_i}}, \tag{7}$$

where mp_i is the number of computational units needed to process an incoming message from server s_i, sp_i is the number of computational units required for dealing with the service invocation on s_i, rp is the number of computational units needed to process the results obtained from service invocations, c_c is the computing capability of the client, and c_{s_i} is the computing capability of server s_i.

With respect to the client-server paradigm, the processing time, P^{cs}, counts the time that the client takes in processing messages replied by all servers, i.e., $\frac{(\sum_{i=1}^{n} mp_i)}{c_c}$, and the obtained results, i.e., $\frac{rp}{c_c}$, and the time that the servers take in tackling service invocations, i.e., $\sum_{i=1}^{n} \frac{sp_i}{c_{s_i}}$. In contrast, as for the mobile agent paradigm, the processing time, P^{ma}, counts the time $\frac{mp_n}{c_c}$ that the client takes in processing the message replied by the last traversed server, including serialization/de-serialization of a mobile agent, encapsulation of the mobile agent as a Java archive file, and storing of the mobile agent in a docking server. In addition, P^{ma} counts the time that the client takes in processing the obtained results, i.e., $\frac{rp}{c_c}$, and the time that the servers take in processing the message coming from the last server that the mobile agent has visited, i.e., $\sum_{i=1}^{n-1} \frac{mp_i}{c_{s_i}}$, and tackling service invocations, i.e., $\sum_{i=1}^{n} \frac{sp_i}{c_{s_i}}$.

The turnaround time is defined as the summation of transmission time and processing time in Definition 4.

Definition 4. *Given the transmission time T^{cs} and T^{ma}, and the process time P^{cs} and P^{ma}, the turnaround time spent in the client server paradigm, denoted as R^{cs}, and the turnaround time spent in the mobile agent paradigm, denoted as R^{ma}, are defined as follows:*

$$R^{cs} = T^{cs} + P^{cs} \tag{8}$$

$$R^{ma} = T^{ma} + P^{ma}. \tag{9}$$

5.2 Experimental Scenario

To setup an experimental environment for performance evaluations, a car accident notification application with a number of services deployed on distributed OSGi platforms is developed.

The car accident notification application exploits mobile agents to deliver five services provided by distributed OSGi platforms. The BEPL-based itinerary that guides mobile agents to access the five services is depicted in Fig. 9. When an accident occurs, a driver can make a request to a mobile agent deployed in his/her car On Board Unit (OBU) to perform the accident notifications on behavior of the driver. The driver can use the OBU to take photos of the collision scene, and activate a mobile agent to trigger the local `GPS` service to obtain the GPS coordinates of the accident location. The mobile agent then carries the photos, GPS information, car profile, and time of accident to report the accident to a traffic information center. What happens next is that it migrates to an OSGi platform providing an `AreaMapping` service to obtain information about the area where the accident occurs. It uses the information to discover the roadside camera service `RoadsideCamera` that operates the roadside surveillance video in the area. The agent then moves to the OSGi platform, where the camera service resides, and triggers it to retrieve the video frames captured the car's trajectory prior to the accident. In the next step, the mobile agent visits the traffic information center and invokes the `AccidentReport` service to report the accident. After the service is invoked by a mobile agent, the screen of the traffic information center displays the map with the location of the accident, and shows the accident photos and roadside surveillance video carried by the mobile agent. The staff of the traffic information center can dispatch rescue and ambulance services afterward to the

```
<MobileAgent>
<itinerary>
<process name="car accident notification">
  <profile homeName="OBU1" id="MobileAgent:1236682460000"/>
  <pointer invoke="home"/>
  <partnerLinks>
    <partnerLink name="home" />
    <partnerLink name="GPSServicePL" />
    <partnerLink name="AreaMappingPL" />
    <partnerLink name="RoadSideCameraServicePL" />
    <partnerLink name="AccidentReportServicePL" />
    <partnerLink name="OBUServicePL" />
  </partnerLinks>
  <variables>
    <variable messageType="ca:emergencyMsg"
    name="EmergencyMsg"/>
    <variable type="xsd:string" name="Area"/>
  </variables>
    <sequence>
      <receive variable="EmergencyMsg" name="home"/>
       ...
      <invoke inputVariable="EmergencyMsg"
        operation="getRoadSideVideo"
        outputVariable="EmergencyMsg" partnerLink=""
        constraints="area=*Area"
        portType="RoadSideCameraService"
        name="invokeRoadSideCameraService"/>
        ...
      <reply operation="playRoadSideVideo"
        partnerLink="OBUServicePL"
        portType="OBUService" variable="EmergencyMsg"
        name="reply-output"/>
    </sequence>
</process>
</itinerary>
</MobileAgent>
```

Fig. 9. A BPEL-based itinerary for the car accident notification application.

scene of the accident. Finally, the mobile agent returns to the initiator's car OBU and invokes the local service to play the retrieved roadside surveillance video on the OBU to show the driver what has happened.

To realize this scenario, we have developed an OBU prototype based on an ARM 11 development reference board with 667-MHz CPU and 128-MB memory, on which the car accident notification application is implemented based on our itinerary-based mobile agents. The OBU is also equipped with a GPS function to get the current GPS coordinates. A Java virtual machine, Java SE for Embedded 6, is installed in the prototypical OBU for laying out an OSGi platform. Knopflerfish 2.3.3, which implements the OSGi R4 Service Platform specification, is adopted to setup OSGi platforms in distributed sites. In this exemplary scenario, several OBU applications are implemented as OSGi bundles, including taking photos of an accident, launching mobile agents, and displaying the roadside surveillance video carried back by mobile agents.

On the traffic information center side, we use a web browser as a user interface to display information of car accident events reported by drivers. When the `AccidentReport` service is invoked by a mobile agent, the service makes a call of `Google Map` APIs to provide a map of the area around the location of the accident, and to show the accident pictures and video.

5.3 Comparison with R-OSGi and SOAP-OSGi

What follows is an experiment conducted to examine MA-OSGi with two related approaches: R-OSGi and SOAP-OSGi, based on the two metrics defined above, namely, network traffic and turnaround time, by actually implementing these three methods for the car accident notification application.

R-OSGi [29] is a distributed middleware built on top of OSGi platform, by which service proxies for OSGi service invocation are generated and deployed in clients to make service remotely accessible. Meanwhile, SOAP-OSGi is implemented by adopting Knopflerfish [16] that provides an Axis bundle to transform OSGi services into web services that can be invoked through SOAP. SOAP-OSGi and R-OSGi are client-server models, whereas MA-OSGi is a mobile agent model. Detailed description of R-OSGi and SOAP-OSGi can be found in Section 2.

To make a fair comparison, two alternations are introduced in the experiments:

- Since R-OSGi and SOAP-OSGi do not support service matchmaking at runtime, we assume that the location of the `RoadsideCamera` service is set to be fixed to ignore the overhead of runtime matchmaking.
- By default, JAR files used to encapsulate mobile agent bundles are compressed, and thus, video frames carried by mobile agents are compressed too. As R-OSGi and SOAP-OSGi do not compress transferred video data, MA-OSGi-wo-C, which is without compression, and MA-OSGi-w-C, which is with compression, are derived from MA-OSGi.

In the experiment, the number of video frames obtained from the roadside camera varies from 50 to 500, and we observe the turnaround time and the network traffic of R-OSGi, SOAP-OSGi, MA-OSGi-wo-C (without compression), MA-OSGi-w-C (with compression) over a wireless network of 256-KB/s bandwidth. The size of each video frame is approximately 12 KB. We repeat the evaluation of these four approaches for each case 10 times and tally the average results as the experiment outcome.

First, we measure the network traffic for each of the four approaches. We begin with the evaluation of the client-side network traffic—the amount of data transferred between the client and the servers (see (1) and (2) in Definition 1). The evaluation result is shown in Fig. 10a, in which MA-OSGi generates less network traffic than the other two, no matter if compression method is adopted by MA-OSGi, or not. As mentioned earlier, MA-OSGi only requires two connections between the client and remote services to finish a task, and therefore, it generates the least network traffic. This experiment result shows that MA-OSGi can significantly reduce the amount of network traffic between mobile devices and servers in a mobile computing environment.

Another evaluation is conducted to take into account the network traffic between servers. Fig. 10b shows the evaluation result of the client-server network traffic together with server-server network traffic (see (1) and (3) in Definition 1). Because there is no network traffic between servers in R-OSGi and SOAP-OSGi, their results are the same as Fig. 10a. In MA-OSGi, the mobile agent mobilizes between servers for service invocations, and consequently generates network traffic between servers.

Because the network bandwidth between a client and servers is far more restricted than the network bandwidth between servers in a mobile computing environment, applying MA-OSGi in mobile applications can reduce more client-server network traffic than R-OSGi and SOAP-OSGi as the size of the transferred data increases.

Fig. 11a shows the transmission time (Definition 2) of the four approaches. In the experiment, the transmission time is

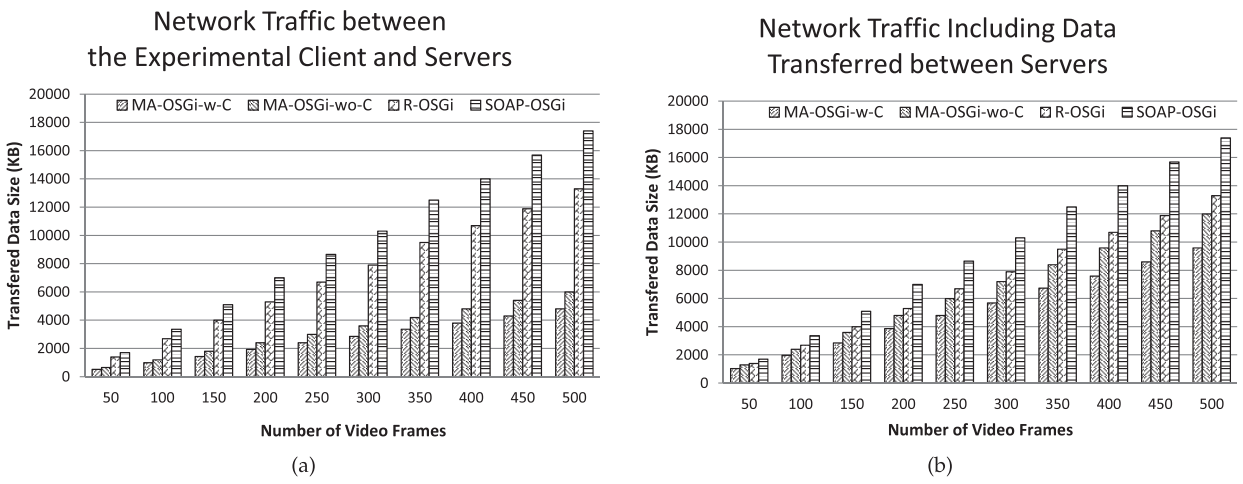

Fig. 10. (a) Network traffic generated between the client and servers; (b) Network traffic including the data transferred between servers.

largely influenced by the data size of the video frames captured by the road-side camera. In the cases of MA-OSGi-w-C and MA-OSGi-wo-C, the video frames carried by the mobile agent only need to be transferred once from the server site of the accident report service to the client side; whereas, in the cases of SOAP-OSGi and R-OSGi, the video frames not only need to be transferred from the server site of the road-side camera service to the client but also need to be transferred subsequently from the client to the server site of accident report service for reporting the accident. Consequently, the experiment results show that MA-OSGi-w-C and MA-OSGi-wo-C require less transmission time than the other two approaches.

Fig. 11b shows the processing time (Definition 3) of the four approaches. It is noted that the processing time of R-OSGi is less than that of MA-OSGi-wo-C by a nearly constant value of 280 milliseconds even with the increases of the number of video frames. Because the influence of data transmission time trumps that of the processing time as the size of the transferred data increases, the turnaround time (namely, the sum of the transmission time and processing time) of MA-OSGi-wo-C and MA-OSGi-w-C are still less than that of R-OSGi and SOAP-OSGi (see Fig. 12).

5.4 Discussion

In general, the dependency between two server-side services in a workflow process can be broken down into four categories with respect to the relationship between their inputs and outputs:

- *Type A*. The output of the first service becomes the input of the second service, and the output is not transferred to the client. The client-server approaches usually take time to transmit the output from the server of the first service (called Server I) to the client, and then to transmit the output from the client to the server of the second service (called Server II) for the invocation of the second service. However, in MA-OSGi, a mobile agent carries the output data of the first service and then mobilizes to server II for the service invocation; and therefore, there is no need to transmit the output data between the client and the servers.

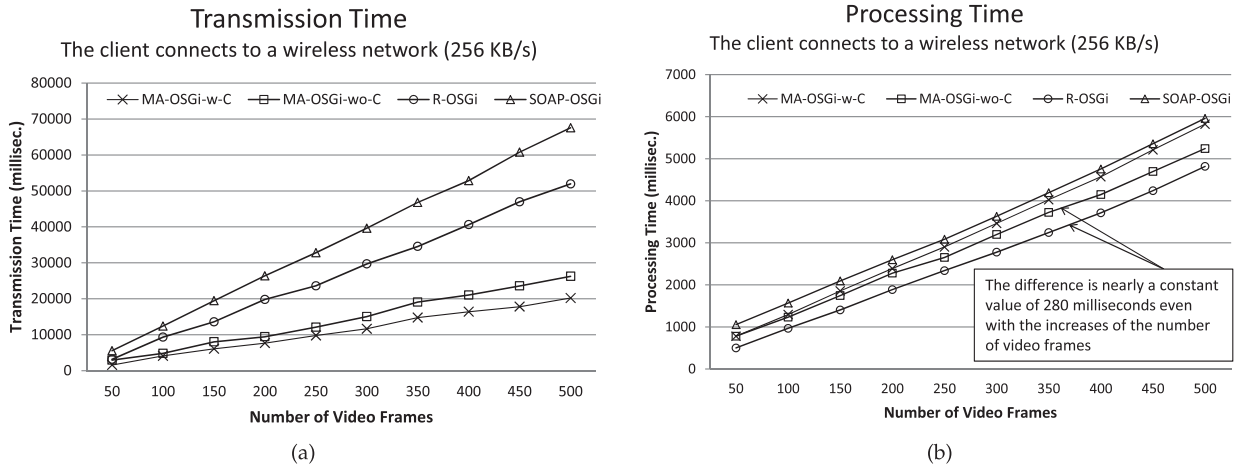

Fig. 11. (a) Transmission time; (b) Processing time.

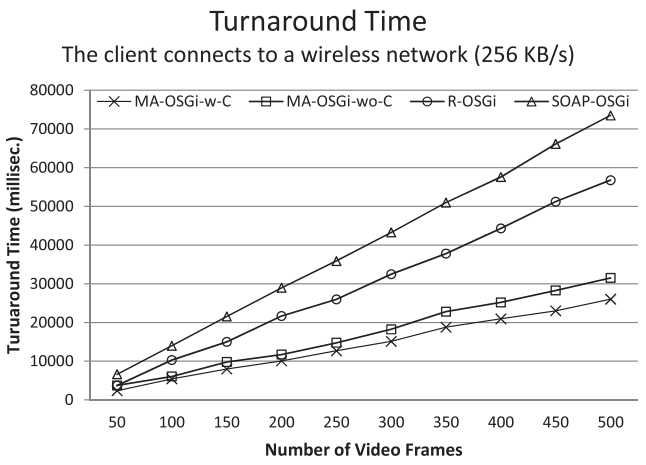

Fig. 12. Turnaround time.

- *Type B*. The output of the first service becomes the input of the second service, and the output is then transferred to the client. Similar to the behavior of type A, the client-server approaches transmit the output data in between the client and the servers. However, in MA-OSGi, a mobile agent with the output data of the first service mobilizes from Server II to the client after the invocation of the second service. There is still no need to transmit the output data from the client to server II.
- *Type C*. The output of the first service is not the input of the second service, and the output is transferred to the client. The client-server approaches transmit the output data from server I to the client, and meanwhile, our MA-OSGi only needs to transmit the output data from server II to the client.
- *Type D*. The output of the first service is not the input of the second service, and the output is not transferred to the client. Both the client-server approaches and MA-OSGi do not need to transmit the output data in between the client and the servers.

With types C and D, the size of the data to be transferred in between the client and the servers in MA-OSGi is about the same as that of the client-server approaches. With types A and B, the MA-OSGi generates less network traffic in between the client and the servers than the client-server approaches. The experiment result shows that our MA-OSGi largely outperforms the client-server approaches on the turnaround time and network traffic as the number of the video frames increases, and the key reason behind is that the output (video frames) of the road-side camera service is not required to be transferred from the client to the server of the accident report service for the service invocation (same as the service dependency of type B). As a result, MA-OSGi is more suitable for the scenarios of workflow processes composed of numerous server-side services of types A and B and with large service output data.

6 Conclusion

This work presents an itinerary-based mobile agent approach to distributed OSGi services, with the following key features:

1. MAEE and mobile agents are developed on top of OSGi platforms to support reduce resource consumptions on mobile devices;
2. Asynchronous communication mechanism is devised to enable reliable mobile agent transmissions among mobile devices;
3. OSGi services and mobile agent itineraries are described by the extended WSDL and BPEL, respectively;
4. A PPN-based opportunistic service matchmaking mechanism is provided to help mobile agents make dynamic adjustment to any changes of context changes.

We also conducted experiments to illustrate the benefits of MA-OSGi by comparing our work with R-OSGi and SOAP-OSGi. Two performance metrics are considered in the experiments: network traffic and turnaround time. The experiment results show that MA-OSGi outperforms the other two approaches with respect to the two metrics, and therefore, MA-OSGi can be considered as a more suitable platform for workflow-based mobile computing applications than R-OSGi and SOAP-OSGi from the perspective of resource consumptions on mobile devices.

As an OSGi bundle is enabled to mobilize among different hosts over networks, how to secure the itinerary and data inside a bundle from being accessed by malicious parties is one of our key future tasks. On providing secure communication environments for the transferring of bundles, we plan to integrate the protocols of MA-OSGi with HTTPS scheme to create secure channels over insecure networks. On providing secure execution environments in which all bundles will be trusted in a controlled manner, the MAEE will be enhanced with the OSGi's code authentication mechanism that can authenticate bundles based on a digitally signing framework.

Acknowledgments

This research was sponsored by the National Science Council (Taiwan) under the grants NSC 100-2631-H-002-035 and NSC 102-2622-E-002-007-CC1.

References

[1] A.R. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R. Singh, "Mobile Agent Programming in Ajanta," *Proc. IEEE 19th Int'l Conf. Distributed Computing Systems,* pp. 190-197, 1999.

[2] J. Baumann, F. Hohl, K. Rothermel, and M. Straßer, "Mole - Concepts of a Mobile Agent System," *World Wide Web,* vol. 1, no. 3, pp. 123-137, 1998.

[3] C. Bäumer, M. Breugst, S. Choy, and T. Magedanz, "Grasshopper - A Universal Agent Platform Based on OMG MASIF and FIPA Standards," *Proc. First Int'l Workshop Mobile Agents for Telecomm. Applications (MATAA99),* pp. 1-18, 1999.

[4] B. Chen, D.D. Linz, and H.H. Cheng, "Xml-Based Agent Communication, Migration and Computation in Mobile Agent Systems," *J. Systems and Software,* vol. 81, no. 8, pp. 1364-1376, 2008.

[5] K. Chiu, M. Govindaraju, and R. Bramley, "Investigating the Limits of SOAP Performance for Scientific Computing," *Proc. IEEE 11th Int'l Symp. High-Performance Distributed Computing,* pp. 246-254, 2002.

[6] P. Dobrev, D. Famolari, C. Kurzke, and B. Miller, "Device and Service Discovery in Home Networks with OSGi," *IEEE Comm. Magazine,* vol. 40, no. 8, pp. 86-92, Aug. 2002.

[7] H. Eikerling and F. Berger, "Design of OSGi Compatible Middleware Components for Mobile Multimedia Applications," *Proc. Protocols and Systems for Interactive Distributed Multimedia Systems, Joint Int'l Workshops Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems (IDMS/PROMS '02),* pp. 80-91, 2002.
[8] A. Ferscha, M. Hechinger, R. Mayrhofer, and R. Oberhauser, "A Light-Weight Component Model for Peer-to-Peer Applications," *Proc. 24th Int'l Conf. Distributed Computing Systems Workshops,* pp. 520-527, 2004.
[9] G. Fortino, A. Garro, and W. Russo, "Achieving Mobile Agent Systems Interoperability through Software Layering," *Information and Software Technology,* vol. 50, no. 4, pp. 322-341, 2008.
[10] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison Wesley, 1995.
[11] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. Software Eng.,* vol. 24, no. 5, pp. 342-361, May 1998.
[12] R.S. Gray, "Agent Tcl: A Transportable Agent System," *Proc. CIKM Workshop Intelligent Information Agents (CIKM '95),* 1995.
[13] O. Gruber, B. Hargrave, J. McAffer, P. Rapicault, and T. Watson, "The Eclipse 3.0 Platform: Adopting OSGi Technology," *IBM Systems J.,* vol. 44, pp. 289-299, 2005.
[14] A. Ibrahim and L. Zhao, "Supporting the OSGi Service Platform with Mobility and Service Distribution in Ubiquitous Home Environments," *The Computer J.,* vol. 52, no. 2, pp. 210-239, 2009.
[15] K. Kang, J. Lee, and H. Choi, "Extended Service Registry for Distributed Computing Support in OSGi Architecture," *Proc. Eighth Int'l Conf. Advanced Comm. Technology (ICACT '06),* pp. 1631-1634, 2006.
[16] Knopflerfish 3, OSGi R4. http://www.knopflerfish.org/, 2013.
[17] M. Kim, Y. Choi, Y. Moon, S. Kim, and O.-C. Kwon, "Design and Implementation of Status Based Application Manager for Telematics," *Proc. Eighth Int'l Conf. Advanced Comm. Technology (ICACT '06),* pp. p1364-1366, 2006.
[18] J. Kiniry and D. Zimmerman, "A Hands-on Look at Java Mobile Agents," *IEEE Internet Computing,* vol. 1, no. 4, pp. 21-30, July/Aug. 1997.
[19] K. Kotay and D. Kotz, "Transportable Agents," *Proc. Third CIKM Workshop Intelligent Information Agents (CIKM '94),* 1994.
[20] J.F. Kurose and K.W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet.* Addison Wesley, 2004.
[21] D.B. Lange and M. Oshima, "Mobile Agents with Java: The Aglet API," *World Wide Web,* vol. 1, no. 3, pp. 111-121, 1998.
[22] D.B. Lange and M. Oshima, "Seven Good Reasons for Mobile Agents," *Comm. ACM,* vol. 42, no. 3, pp. 88-89, 1999.
[23] J. Lee, Y.-Y. Lin, S.-P. Ma, and S.-J. Lee, "BPEL Extensions to User-Interactive Service Delivery," *J. Information Science and Eng.,* vol. 255, no. 5, pp. 1427-1445, 2009.
[24] J. Lee, K.F.R. Liu, and W. Chiang, "Modeling Uncertainty Reasoning with Possibilistic Petri Nets," *IEEE Trans. Systems, Man, and Cybernetics, Part B: Cybernetics,* vol. 33, no. 2, pp. 214-224, Apr. 2003.
[25] J. Lee, K.F.R. Liu, Y.-C. Wang, and W. Chiang, "Possibilistic Petri Nets as a Basis for Agent Service Description Language," *Fuzzy Sets and Systems,* vol. 144, no. 1, pp. 105-126, 2004.
[26] J. Lee, Y.-C. Wang, C.-L. Wu, S.-J. Lee, S.-P. Ma, and W.-Y. Deng, "A Possibilistic Petri-Nets-Based Service Matchmaker for Multi-Agent Systems," *Int'l J. Fuzzy Systems,* vol. 7, no. 4, pp. 199-213, 2005.
[27] OSGi Alliance, "OSGi Service Platform, Core Specification, Release 4," http://www.osgi.org/, 2013.
[28] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents," *Proc. First Int'l Workshop Mobile Agents (MA '97),* pp. 50-61, 1997.
[29] J. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed Applications through Software Modularization," *Proc. ACM/IFIP/USENIX Eighth Int'l Middleware Conf. (Middleware '07),* pp. 1-20, 2007.
[30] J.S. Rellermeyer, G. Alonso, and T. Roscoe, "Building, Deploying, and Monitoring Distributed Applications with Eclipse and R-OSGi," *Proc. OOPSLA Workshop Eclipse Technology eXchange (Eclipse '07),* pp. 50-54, 2007.
[31] R.S. Hall and H. Cervantes, "Challenges in Building Service-Oriented Applications for OSGi," *IEEE Comm. Magazine,* vol. 42, no. 5, pp. 144-149, May 2004.
[32] The Eclipse Foundation, "Eclipse Communication Framework," http://www.eclipse.org/ecf/, 2013.
[33] OASIS Standard, UDDI Version 3, 2004. http://uddi.org/pubs/uddi_v3.htm, 2013.
[34] C. Williamson, "Internet Traffic Measurement," *IEEE Internet Computing,* vol. 5, no. 6, pp. 70-74, Nov./Dec. 2001.
[35] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet, "Concordia: An Infrastructure for Collaborating Mobile Agents," *Proc. First Int'l Workshop Mobile Agents (MA '97),* pp. 86-97, 1997.
[36] C.-L. Wu, C.-F. Liao, and L.-C. Fu, "Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology," *IEEE Trans. Systems, Man, and Cybernetics, Part C: Applications and Rev.,* vol. 37, no. 2, pp. 193-205, Mar. 2007.
[37] D. Xu, J. Yin, Y. Deng, and J. Ding, "A Formal Architectural Model for Logical Agent Mobility," *IEEE Trans. Software Eng.,* vol. 29, no. 1, pp. 31-45, Jan. 2003.
[38] I. Yamasaki, K. Yata, H. Maeomichi, A. Tsutsui, and R. Kawamura, "Implementation of a Distributed Network Middleware 'CSC' on OSGi Frameworks," *Proc. IEEE Second Consumer Comm. and Networking Conf. (CCNC '05),* pp. 150-155, 2005.

Jonathan Lee is a professor in the Department of Computer Science and Information Engineering at National Taiwan University (NTU) in Taiwan. He was the department chairman from 1999 to 2002 and was the director of Computer Center at National Central University from 2006 to 2012. His research interests include software engineering, service-oriented computing, and software engineering with computational intelligence. He has authored more than 100 journal articles and refereed conference papers. He was awarded IBM Shared University Research Award (2010), CIEE Electrical Engineering Outstanding Professor Award, NCU Distinguished Professor Award, (2006-2013), and NCU Distinguished Research Award (2004). He also served as the program chairs of the 12th Asia-Pacific Software Engineering Conference (APSEC 2005) and the 8th International Fuzzy Systems Association World Congress (IFSA 1999). He received his PhD in computer science from Texas A&M University in 1993.

Shin-Jie Lee is an assistant professor at the Computer and Network Center at National Cheng Kung University (NCKU) in Taiwan and holds joint appointments from the Department of Computer Science and Information Engineering at NCKU. His current research interests include software engineering, agent-based software engineering and service-oriented computing. He received his PhD in computer science and information engineering from National Central University in Taiwan in 2007.

Hsi-Min Chen received the PhD degree in computer science and information engineering from National Central University, Taiwan, in 2010. He has been a software engineer at Institute of Information Science, Academia Sinica, Taiwan, since 2001. His primary research interest includes grid computing, service-oriented software technology, software engineering, and computer supported cooperative work.

Kuo-Hsun Hsu received the PhD degree in computer science and information engineering from National Central University, Taiwan, in 2003. He is currently an assistant professor of the Department of Computer and Information Science, National Taichung University. His research interests include software engineering, requirement engineering, software architecture, service-orient architecture, and CMMI.