Advance Access publication on June 17, 2008
doi:10.1093/comjnl/bxn032

Supporting the OSGi Service Platform with Mobility and Service Distribution in Ubiquitous Home Environments

Abdelgadir Ibrahim* and Liping Zhao

School of Computer Science, The University of Manchester, Kiburn Building, Oxford Road, Manchester M13 9PL, UK
*Corresponding author: *abdelgadir.ibrahim@manchester.ac.uk*

The OSGi service specification defines an open service platform for service delivery, composition and execution in networked environments. The specification, however, is limited to a single java virtual machine (JVM) and does not define the distribution and mobility of services across different OSGi platforms and devices. This paper first revisits the fundamentals of OSGi service distribution, clarifies and defines a terminology for OSGi service mobility and distribution. It then proposes to extend the current OSGi platform with service distribution and service mobility that aim to support three important requirements on ubiquitous applications, namely, spontaneous interoperability, mobility and software adaptability. The paper demonstrates these extensions through several prototype implementations. These extensions are supported through a common framework, which targets at ubiquitous environments and aims to facilitate the construction of OSGi applications that span multiple OSGi platforms, multiple JVMs and multiple devices. In addition, the proposed framework offers two special features: First, it supports automatic contextual management through a virtual global shared space whose content is automatically and dynamically adjusted to reflect the changes in the system and the mobile environment; Second, it supports different OSGi bundle and service mobility paradigms. The proposed framework blurs the distinction between local and remote services, where remote services can be accessed as if they were local, which greatly simplifies application development. We believe existing OSGi platform distribution solutions can also be supported by this framework.

Keywords: service-oriented systems; distributed systems; ubiquitous environments; mobility; OSGi

Received 6 July 2007; revised 27 March 2008

1. INTRODUCTION

Ubiquitous computing, also known as pervasive computing [1–3], describes a trend towards environments in which information and communication technology is integrated into the environment through interactions with everyday devices. Other terms that describe this computational paradigm include 'ambient intelligence', and 'everywhere'. In such an environment, a user interacts with multiple, connected consumer devices, instead of one single device. Thus, computation is distributed among different connected devices in the user's environment. The goal of ubiquitous computing is to create environments that are characterized by unobtrusive and always available connectivity. Ubiquitous environments are also characterized by mobility of users and devices, intermittent network connections, diverse network protocols and dynamic introduction and removal of devices. Together, these characteristics complicate the development of ubiquitous applications. This trend of ubiquitous computing is largely fuelled by advances in consumer electronics and wireless technologies.

Requirements on pervasive computing have been discussed by many authors (see for example, [2–5]). According to these authors, *spontaneous interoperability*, *mobility* and *adaptability* are three interconnected requirements that are important to pervasive computing. These three requirements are described below.

- *Spontaneous interoperability*. Proliferation of networked environments has led to increasing requirement to connect devices in order to provide added value services such as the ability to instruct a sound system to play a specific song that is stored in a PC [6]. However, connecting devices is difficult due to the heterogeneity of devices, networking technologies and adopted communication protocols [7, 8]. End users, on the other hand, are only interested in *services* that span device boundaries. This heterogeneity, especially in the connected home domain, is likely to hold for the foreseeable future.

- *Mobility*. Mobility can be classified into *physical* mobility (the device itself physically moves from one location to another), *personal* mobility (the device may be fixed but it is the user who moves between different locations), *virtual* mobility (the application is aware of the distributed nature of the execution environment and is able to locate and access such remote resources and services) and *logical* mobility (it is possible to withdraw a service in one location and reoffer it in a different location) [2]. While mobility is an important requirement on pervasive computing, it should not affect the correct functioning of the applications; applications should be able to adapt accordingly.
- *Adaptability*. The dynamic nature of ubiquitous environments, where devices may be dynamically added or removed, users or devices may move and network connections may dynamically come and go, calls for dynamic software architectures that facilitate *self-organization* of the software accordingly. A taxonomy of adaptation strategies, their advantages and disadvantages is given in [9].

To date, most ubiquitous applications are supported by the service-oriented paradigm [10, 11]. Technologies such as UPnP[1] and Jini[2] have been used to build ubiquitous applications. However, although these technologies are meant to support interoperability of devices, in reality, they only allow devices of the same protocol to interoperate. For example, an UPnP device cannot directly communicate with a Jini device.

An alternative, promising service-oriented technology for ubiquitous applications is the OSGi service platform [12]. The OSGi specification is designed to complement and enhance virtually all residential networking standards and initiatives including Bluetooth^{TM}, CAL^{TM}, CEBus^{TM}, HAVi^{TM}, HomePlug^{TM}, HomeRF^{TM}, Jini^{TM} and UPnP^{TM}. OSGi technology offers many benefits to application development, including: security, modularity, decoupling, portability, lightweight, architecture openness, platform extensibility, dynamicity, potential for remote device management, support for multiple networking technologies, multiple devices and multiple vendors. Consequently, the OSGi platform is becoming a *universal* middleware for different applications in a variety of domains ranging from consumer electronics to enterprise server applications. Many recent research efforts in pervasive computing have adopted the OSGi platform as an underlying middleware. For example, the application of the OSGi platform as a middleware for home environments has been described by many authors including [6–8, 13–15]. A home network architecture and an overview of technologies, including OSGi technology, that aim to solve the interoperability problem are given in [16].

However, current implementations of OSGi technology only provide limited support to the above three pervasive requirements. For example, they are restricted to a *single* java virtual machine (JVM) and subsequently do not support remote service invocations common in distributed systems [17]. Different extensions to the OSGi platform have been proposed, but to our knowledge, they are still limited; their support for adaptability and spontaneous interactions is either limited or lacking. In addition, these extensions mainly target fixed network settings and are not suited for pervasive environments. The challenge in supporting adaptability and spontaneous interactions lies in integrating such support with the existing OSGi application model that is characterized by strong modularity.

This paper proposes two extensions to the current OSGi platform, which collectively support the aforementioned three requirements. These two extensions are described below:

- *Support for spontaneous interoperability in mobile environments*. Pervasive devices should be able to interact spontaneously as they become connected which rules out centralized architectures that many authors, e.g. [14], have considered inadequate for home networks. Justified by the increasing availability of OSGi-enabled devices, we propose a peer-to-peer (P2P) model [18], where various OSGi-enabled devices are able to spontaneously interact by offering and using services from each other. Consequently, this extension supports the OSGi platform with service distribution across OSGi platforms and devices while accommodating such P2P interactions as well as device mobility.

  Most current OSGi platform distribution extensions use communication models that are based on remote procedure calls (RPCs) [19], e.g. Java remote method invocation RMI. These communication models are inadequate for pervasive environments due to the characteristics of mobility and intermittent connectivity. Another limitation of using communication technologies such as Java RMI is that it breaks OSGi platform modularity. Other OSGi platform distribution extensions which have developed their proprietary communication protocols do not support spontaneous interactions. In contrast, our extension can be configured to use the publish–subscribe interaction style [20], which is considered as a better alternative for communication in pervasive environments. In addition, our proposed extension supports spontaneous interactions while observing OSGi modularity.

[1]www.upnp.org.
[2]www.jini.org.

- *Support for adaptability*. In order to support software adaptability, the OSGi platform must also support *logical* mobility, which is currently lacking. This extension supports the OSGi platform with bundle and service mobility across OSGi platforms and devices. Logical mobility as manifested by mobile software agents and other mobile abstractions is considered to provide many benefits to software applications (see for example, [21, 22]).

These extensions are supported through a common framework, which targets at ubiquitous environments and aims to facilitate the construction of OSGi applications that span multiple OSGi platforms, multiple JVMs and multiple devices. This framework is also applicable to non-pervasive environments where OSGi technology is used. The idea is to have an abstract mobility and distribution model from which different *instantiations* can be created that target different application domains. In the reminder of the paper, we refer to this framework backed by the abstract mobility and distribution model as the *conceptual framework*. This proposed conceptual framework offers two special features: First, it supports automatic contextual management through a virtual global shared space whose content is automatically and dynamically adjusted to reflect changes in the system and the mobile environment; Second, it supports different OSGi bundle and service mobility paradigms. The proposed conceptual framework blurs the distinction between local and remote services, where remote services can be accessed as if they were local, which greatly simplifies application development. We believe existing OSGi distribution solutions can also be supported by this framework. Another contribution of this paper is a characterization of spontaneous interactions and logical mobility in the context of the OSGi platform. Finally, a secondary outcome of this work, is a survey of OSGi service distribution.

The rest of this paper is organized as follows. Section 2 provides the background to the paper. Sections 3 and 4 describe the proposed conceptual framework and its architecture, respectively. The design and implementation of the D-OSGi mobility extension which realizes the proposed conceptual framework is described in Section 5. The work is evaluated in Section 6, whereas Section 7 discusses related work. Finally, Section 8 concludes the paper.

2. BACKGROUND

2.1. Overview of the OSGi middleware

The OSGi alliance has defined the OSGi specification for a service platform that supports the delivery of managed services to networked environments such as homes. The initial focus of the OSGi specification was the market of home service gateways that link the home network to external service providers who can then provide remotely managed services to the home through the gateway. Typical such services include home security and home health care monitoring. However, applications of the OSGi technology proved to be much wider than service gateways and consequently the OSGi technology scope was further extended to include *any* networked environment, e.g. cars, that provide a mix of embedded devices and the need to deploy services for those devices. In other words, gateway is only one of OSGi's application areas which currently include industrial automation, mobile environments, automotive, desktop and server applications.

The OSGi service platform consists of two elements: the OSGi framework and a set of *standard* service definitions. The OSGi framework is a lightweight Java-based container (fully J2ME-compatible) for deploying and executing *service-oriented* applications. The OSGi framework defines a component model, a services registry and provides the runtime environment for handling the interactions between services and between components. In addition, the OSGi framework supports the remote management of the entire application life cycle including dynamic on-the-device software deployment and extension. The OSGi standard service definitions are optional and can be implemented by different vendors for inclusion in any given solution. These standard services include among many others: an HTTP service, a logging service, an XML parsing service, a UPnP device service and a configuration admin service. As the specification states 'The primary goal of the OSGI service framework is to use the Java programming language platform independence and dynamic code loading capability to make development and dynamic deployment of applications for small memory devices easier'.

The OSGi application model combines the two approaches of component-orientation and service-orientation resulting in what is known as a service-oriented component model [23, 24]. In this model, a software application is viewed as a set of collaborating components that provide and use services to and from each other, respectively. Component collaboration follows the service-oriented interaction pattern, where services provided by components are published into a registry. Client components can then dynamically discover and bind to those published services.

A component in the OSGi service platform is known as a *bundle*. Bundles are the unit of delivery and deployment. A bundle is represented as a Java Archive File (JAR) file that contains all resources required for the operation of the bundle including specifications of its dependencies on the environment and on other bundles. The OSGi platform enables multiple applications realized as bundles to share the same JVM with full support for modularity, where bundles and applications are isolated from each other. A unique and innovative characteristic of bundles is their ability to cooperate and share Java packages, i.e. code, with each other. The OSGi service platform defines the mechanism for code sharing and provides for its automatic management; a process that is known as package dependency resolving. A package can be *exported* by a bundle, thus making the package available for use by other bundles. Conversely, an exported package can be imported by other bundles thus making the package accessible to the importing bundles. In addition to package sharing via the import/export mechanism, bundles can also contain private packages that are hidden from other bundles. OSGi platform uses the Java *class loader* as the mechanism to enable the sharing and hiding of packages and classes. Each bundle is assigned a separate class loader that loads classes using well-defined rules. This results in multiple class spaces for the different bundles. A class space can be defined as all reachable classes from a bundle's class loader.

The OSGi platform supports the dynamic download, deployment and management of bundles within a *single* JVM. It provides support for installing, uninstalling, activating (starting), deactivating (stopping), updating and refreshing of bundles. All these activities can be performed either locally or remotely. Together, these activities also define the bundle's life cycle. A bundle can only be activated if its package dependencies are resolved. When activated, a bundle can then offer an arbitrary number of services by publishing them through the OSGi registry. Other active bundles can dynamically discover and bind to those published services. An active bundle can also, at any time, withdraw a previously published service. Clients using the withdrawn service are automatically notified and must adapt to such change in service availability, e.g. by searching and binding to an alternative service. When a bundle is deactivated, all its offered services are automatically withdrawn and must also release any other services it is currently using. Once deactivated, a bundle re-enters the resolved state and can then be either uninstalled or reactivated. Several commercial and open source OSGi platform implementations exist including Apache Felix[3], Eclipse Equinox[4] Knopflerfish[5] and Concierge[6]

2.2. An introduction to code mobility

Code mobility is defined informally as 'the capability to dynamically change the bindings between code fragments and the location where they are executed [22]'. In other words, code mobility provides the capability to move code across nodes in a network. In relation to logical mobility, code mobility can be considered at a lower layer on top of which logical mobility is built. Logical mobility deals with abstractions at the application level such as services, agents and their used resources.

Fuggetta *et al*. [22] introduce the concepts of *computational environment* (*CE*), *execution unit* (*EU*) and *State*. CEs, which are also known as locations or places [25], and as agent servers, are abstractions for entities that provide the relocation capabilities and maintain the identities of the hosts in which they reside. EUs, which are also known as agents [25, 26], represent the entities which are hosted by the CEs. Resources are the entities that are needed for EUs to perform their functions. State comprises a *data space* and an *execution state* [22, 25]. The former represents the configuration data required by the EU, whereas the latter represents the runtime data of the EU such as its runtime execution stack. In mobile code systems, all of the code, resources, execution state and the data space of an EU can be relocated to a different CE across the network.

The authors in [22, 25] distinguish between *strong* and *weak* mobility. In strong mobility, all the code, data space and execution state of the EU can be relocated to a different node where execution is resumed. In weak mobility, usually only code can be transferred. Weak mobility may also involve the transfer of the EU's initialization or configuration data but the execution state is not transferred. Subsequently, weak mobility can to some extend mimic strong mobility by saving, prior to relocation, all data that is needed to resume execution from the point at which execution had stopped.

Migration and *remote cloning* are the two mechanisms for realizing strong and weak mobility [22]. When migrating an EU, it is first suspended if applicable, and physically disconnected and transferred before execution is started or resumed at the destination CE. Remote cloning involves the spawning of a *copy* of the EU on the remote CE.

When an EU is being transferred to a remote CE, its code can either be fetched by the destination CE from the source CE, or alternatively the code can be shipped by the source CE to the destination CE—concept of *direction of transfer* [22, 25]. Furthermore, the transferred code can be either self-contained or a code fragment [22]. Self-contained code can be used to independently instantiate and execute the EU at destination. Codes fragments must be linked with other codes residing at the destination CE before they can be used.

Fuggetta *et al.* [22] model a resource as a triple (I, V, T), where 'I' is the resource identifier, 'V' is its value and 'T' is its type. Resource types determine the transferability of the resource. For example, a resource of type *hardware device* is usually not transferable while a resource of type *information* is transferable. Even if a resource is transferable, it can be designated as either fixed (disabled transferability) or free (enabled transferability). The transferability status (fixed/free) of transferable resources is determined by the developer on the basis of application requirements such as performance considerations and the nature of the resource. For example, data resources deemed sensitive or confidential may be marked as fixed.

When an EU is transferred (either by migration or remote cloning), the resources it uses may also need to be transferred along with it. This depends on the nature of the resource and the type of link (binding) between the EU and the resource [22].

[3]http://felix.apache.org.
[4]www.eclipse.org/equinox/.
[5]www.knopflerfish.org/.
[6]http://concierge.sourceforge.net/.

In addition, different applications may have different requirements with respect to resource transferring. There are essentially three possible types of bindings between an EU and a resource: *by identifier*, *by value* and *by type* [22]. Binding by value indicates that the EU always requires the same resource instance for its execution. Binding by identifier indicates that only the type and value of the resource matters. Binding by type indicates that only the type of the resource matters (irrespective of the value).

Fuggetta *et al.* [22] summarize the problems of EU migration as follows:

- *Resource relocation*. The question is whether and how the required resources are migrated?
- *Binding reconfiguration*. The question is how can the binding between the EU and the resource be re-established after the EU is migrated?

Depending on the nature of the binding, Fuggetta *et al.* [22] identify the following strategies:

- *Migration by move*. The resource is transferred along with the EU. In other words, the binding between the EU and the resource is not modified. However, this solution can only be employed when the resource is both transferable and free.
- *Use of network references*. The resource is not migrated along with the EU. At destination, the EU establishes a remote reference back to the resource residing in the source CE. This solution is useful when the resource is either not transferable or is fixed. However, performance issues and potential network failures may affect the operations of the EU if this strategy is used.
- *Migration by copy*. In this case, a *copy* of the resource is packaged along with the EU. At destination, the EU rebinds to this copy.
- *Rebinding*. When the EU is migrated, it simply searches for a resource of the same type. If found, the EU rebinds to this found resource.

At the application design level, Fuggetta *et al.* [22] distinguish between: *code components* (the know-how), *computational components* and *resource components* (data or devices). Based on these concepts, the authors in [22, 27, 28] identify the following paradigms of mobile applications:

- *Client–Server*. Clients request the services offered by a potentially remote server. The server hosts both the know-how and the resources required for providing the service. In addition, it is the server who executes the service according to the know-how. Client–server applications are typically implemented using some form of RPCs.
- *Remote evaluation* (*REV*). A computational component may know how to perform a particular task but lacks the resources. In this case, it may choose to migrate to a remote location where the resources are thought to be available. Once there, the computational component performs the task and then delivers (or brings back) the results to the original location.
- *Code on demand* (*COD*). A computational component may have access to resources required for a specific task but lacks the know-how to manipulate these resources. In this case, it may request such know-how (the code) from a remote location which is then delivered.
- *Mobile agents* (*MAs*). Griss and Pour [29] define an agent as ‘a proactive software component that interacts with its environment and other agents as a surrogate for its user, and reacts to significant changes in the environment’. A distinction is often made between strong and weak agents (see, for example, [25]). Both agent types are characterized by *autonomy*, *collaboration*, *persistence* and *mobility* [29, 30]. In addition, strong agents are also *knowledgeable* and *adaptable*. In order for agents to be able to communicate and understand each other, they must adhere to a common interaction standard [29]. Several agent communication languages (ACLs) have been proposed in the literature that provide a standard mechanism for information exchange between agents. A description of the concepts underpinning ACLs and an overview of some ACLs are given in [31]. Further details about agent concepts, agent design and applications of agent technology is provided in [29, 30]. Additional references for mobile and software agents organized by topic can be found in [32].

A computational component (i.e., an agent) may possess the know-how and some of the required resources. It starts execution on the source location. When additional resources are required, it migrates along with the code, data space, execution state and the intermediate results to a remote location where the extra resources are available. Once there, the computational component simply resumes executions. An MA paradigm can be modelled using either REV or COD. For example, REV may be used to model MAs that initiate migration to remote nodes.

Two criteria that differentiate between the above mobile application paradigms are the *initiator* of interaction or mobility and the *direction of transfer*. In terms of establishing a binding between a client and a server, it is either the server who *pushes* its services (or location information) to clients or alternatively it is the clients who search for a suitable server from which required services can be used (*pull*). REV is characterized by push style of mobility, where a computational unit is pushed to a remote node where execution takes place. COD is characterized by pull style of mobility, where a required computational unit is requested from remote nodes which is then delivered.

It must be noted that no single paradigm is optimal for all applications [22, 27, 28]. Therefore, these different paradigms should be evaluated by developers on application-by-application basis according to application requirements. For example, in order to reduce network traffic in an application that computes a summary from data obtained from large remote database, it may be best to apply an REV paradigm wherein the computational unit moves into the remote server next to the large database in order to compute the summary at the server and thus avoid sending large amounts of data over the network. Furthermore, the actual technology to implement the chosen design paradigm must be evaluated on individual application basis because some technologies are more suited for certain design paradigms [22]. Baldi and Picco [33] provide an evaluation of these different mobile code design paradigms in the context of network management and identify the conditions under which they should be used. Further details about these design paradigms and their trade-offs are given in [22, 27, 33].

Finally, a number of issues arise when using code mobility which include:

- *Security*. How can the transferred unit be safely integrated in the receiving node. A mobile unit may pose a security threat to the receiving node or alternatively it may be faulty in which case it could disrupt the operation of the receiving node.
- *Compatibility*. Unless the nodes have compatible architectures with respect to the used CE, operating system and hardware, a mobile unit may not work in the receiving node. For example, a mobile unit may have some dependencies on a lower-layer functionality that is provided by the operating system. Such mobile unit may not work when it is moved to a node that uses a different operating system. The virtual machine concept, e.g. JVM, is often used to address this issue of compatibility.

In this paper, we do not address the security issue directly but instead use the security mechanisms inherent in the OSGi platform. In addition, the paper focuses on OSGi bundle and service mobility across OSGi-enabled nodes. Therefore, the compatibility issue does not arise in terms of integrating the received mobile unit in the destination node. However, the mobile bundle or service may still fail to resolve if its dependencies on the underlying platform and other entities are not satisfied at destination.

3. A CONCEPTUAL FRAMEWORK FOR OSGI MOBILITY AND DISTRIBUTION

This section describes a conceptual framework for *weak* mobility of OSGi bundles and services across OSGi nodes. This framework adapts the concept of global virtual data structures (GVDSs) [34] for OSGi mobility and service distribution. GVDSs describe a meta-model for coordination in mobile environments that is characterized by the following [34]:

- *Distributed*. Each node owns and stores parts of the data structure.
- *Constructive*. For any given node, its local view of the GVDS is formed by combining the individual data structures residing in all reachable nodes.
- *Scope*. Operations performed on data structures residing in remote nodes appear to be local to the invoking client.

Our choice of weak mobility is motivated by the following observations. First, as stated by Baude *et al.* [35], ‘there exists no implementation of strong migration in Java that does not break the Java model or require user instrumentation of the code’. Specifically, supporting strong migration in OSGi platform may not be possible because bundle activation can only be done via the `start(BundleContext bc)` method which, as the name implies, will restart the bundle causing the loss of its execution state. In addition, a bundle’s state may depend on other bundles’ states which further complicates strong migration. Our choice of a weak mobility model is further influenced by two factors: (1) At the application level, it is possible to some extent mimic strong mobility by explicitly saving, prior to relocation, all data that is needed by the bundle or service to resume execution in the destination node. (2) Many applications do not need support for strong mobility where weak mobility will suffice.

3.1. Terminology for OSGi mobility and distribution

The process of supporting OSGi bundle and service mobility across OSGi framework instances can be considered in its roots as an exercise in *code mobility*. This section maps the above mobility terminology to OSGi concepts and entities.

An OSGi framework represents the CE, whereas bundles represent the EUs. A bundle may use a set of resources which can include, for example, required bundles, imported packages, native libraries, required execution environment and other contextual dependencies. Configuration information can be considered as the bundle’s data space. Basically, a resource is any entity required by the bundle in order for the bundle to be successfully deployed and executed. The aim of the proposed conceptual framework is to support relocation of bundles and services across OSGi framework instances. The conceptual framework defines a *node* as an OSGi framework instance. This means that multiple nodes could be hosted in the same physical device. The proposed conceptual framework supports OSGi service distribution and mobility across OSGi nodes independent of the system or network configuration.

Migrating a bundle means that it is uninstalled from the current node, physically transferred to a remote node where it is reinstalled and started. Migration can be either stateless or stateful. Bundle cloning means a *copy* of the bundle is installed and started in a different node. For remote cloning, a bundle can either be replicated or non-replicated.

Replication means that the original bundle remains active in the source node. Note that although the current OSGi specification defines mechanisms for the download of bundles from remote locations, such download can only be done in a single direction (fetching).

At a different level of granularity, it may be required to support the mobility of individual services as opposed to whole bundles. We draw a logical distinction between the two concepts of service *distribution* and service *mobility*. Service distribution corresponds to a client–server interaction paradigm, whereas service mobility corresponds to either COD, REV or MA paradigm. Service migration means the service is withdrawn from the current framework and re-offered in a different node. Remote service cloning means a copy of the service is offered with a different node (either replicated or non-replicated).

Bundles and services can be transferable or non-transferable. For transferable bundles and services, they can be further designated as either free or fixed.

At the application design level, a client–server paradigm means that the bundle or service and all required resources are hosted by the source node. Clients in remote nodes can access the services using a form of RPCs (irrespective of the actual implementation technology).

When REV is employed, the bundle or service is physically migrated or cloned in a remote node where execution takes place. Results can then be delivered, or brought back along with the mobile entity, to the source node.

A COD design paradigm means that the bundle or service can request the download of required entities (bundle or service) from a remote node. The downloaded entity can then be linked with existing code in the destination node and executed.

An MA design paradigm means that it is possible for a bundle or service to start execution in one framework instance and finish it in a different instance.

Implementing remote invocations in a client–server paradigm implies that the formal parameters of remote services must be serializable which raises the question ‘How to reconstruct (deserialize) the invocation arguments (or return values) when they are received in the provider or client nodes, respectively’. In other words, ‘How to locate the classes required for deserializing the received invocation arguments or return values’. Note that method arguments and return values can be one of three types: *primitive*, *standard* (e.g. `java.*`) or *non-standard* (all other types). A non-standard type could have been imported from a bundle which is not available (or available but not accessible) to the receiving entity in the receiving node. In this case, it is not possible to reconstruct the arguments or return values from the serialized stream. Also note that this problem only concerns non-standard types. For standard types, the default OSGi class loading delegation process will eventually use the parent/system class loader to locate the required types. Consequently, in terms of resource availability and resource binding, we distinguish between the following scenarios.

Scenario 1. *A bundle or service is migrated, distributed (for services) or cloned to a remote node but required resources, e.g. declared dependencies, are unavailable in the destination node*.

Because such dependencies are considered as environmental dependencies that must be resolved at the OSGi module layer, migration in this case should fail unless the dependencies are satisfied. A mobility extension may provide support to dynamically identify, locate and install such unsatisfied dependencies, e.g. using the Apache Felix bundle repository bundle[7] (formerly known as the Oscar Bundle Repository Bundle), which provides an OSGi service for dynamically deploying repository bundles and the transitive closure of their deployment dependencies into an executing OSGi framework. In addition, the latest release 4.1 of the OSGi specification introduces a standard Deployment Admin Service to assist in deployment of bundles and related resources. The Deployment Admin service enables the management of a bundle life cycle together with its required resources as a *single unit*. For example, using the Deployment Admin service, it is possible to install/update a bundle, set up configuration objects and configure permission objects all in a single atomic transaction. A deployment package groups a set of resources and actions that must be treated as a single unit. A mobility extension may therefore utilize deployment packages to integrate mobile entities together with their interlinked resources at the destination node.

Scenario 2. *Remote invocation of a method defined by a distributed service requires a resource that is available to the calling service but not the called service*.

In order not to violate the OSGi application model and to link to its resource and class loading delegation model, required resources should be fetched through the corresponding bundle. In this scenario, a mobility extension should dynamically attempt to fetch and download the resource (if transferable) from the bundle of the calling service (which may in turn link to other bundles in the client node).

Scenario 3. *Remote invocation of a method defined by a distributed service requires a resource that is available to the called service but not the calling service. For example, when the type of a return argument is inaccessible to the calling service*.

Again, such resource should be dynamically fetched from the bundle of the called service (which may then delegate to other bundles in the server node).

As mentioned previously, support for the different client–server, REV, COD and MA mobility strategies depends on the nature of the application, its domain and its requirements. Hence, not all the above mobility strategies may be required

[7]www.felix.apache.org.

for all applications and consequently may not be supported by a specific mobility extension implementation.

3.2. Requirements and overview

We have identified the following requirements on the proposed conceptual framework and its instantiations:

- *Spontaneous interactions*. It should be possible to federate a set of nodes spontaneously. No pre-configuration or special provisions other than the mobility extension should be needed.
- *Flexibility*. The OSGi platform is being used in a variety of domains, network models and devices ranging from mobile devices to enterprise servers. Therefore, the solution should be lightweight and flexible so that it can be used in all areas and devices supported by the OSGi platform, and be independent of the adopted network model and communication protocol.
- *Mobility paradigms*. As discussed previously, different applications will use different mobility paradigms. An OSGi mobility extension should therefore support all the aforementioned mobility paradigms. Different applications can then use the most suitable mobility paradigm for their needs.

The proposed conceptual framework which is illustrated in Fig. 1 provides a flexible solution to OSGi bundle and service mobility that aims to address all the aforementioned application paradigms of client-server, REV, COD and MA. A mobility extension manifested as a *mobility bundle* is used to realize bundle and service mobility as well as service distribution across OSGi nodes. The following sections provide a description of elements comprizing this conceptual framework.

3.3. Connectivity and transient sharing

In adapting the concepts of GVDS to OSGi distribution and mobility, the proposed conceptual framework considers an OSGi registry as a structure whose content (services) will be shared with all the available structures (other OSGi registries). When two or more nodes become *connected* either directly or transitively, their local services are *transparently* shared between all the connected nodes. Only services that have been *explicitly* marked for remote access are shared. Conceptually, transient sharing results in the formation of a logical group across the connected nodes that represents a *logical federated* OSGi registry. Such federated registry allows bundles and services residing in different OSGi nodes to interact as if they were all residing in the same node. Management of connectivity and transient sharing is transparent to individual bundles and services. Bundles and services only have to interact with their local registry if they wish to search for remote services, invoke those remote services, find about reachable nodes (i.e. connected nodes) and to request migration to a reachable node.

The proposed conceptual framework considers connectivity at a high level of abstraction. Two nodes are considered connected, and therefore are able to form a logical group if they become within communication range and are able to communicate. Note that it is possible for two nodes to be within communication range but unable to communicate; for example, when the cost of communication is high or when the quality of communication is low. The meaning of connectivity and the interpretation of the clause 'within communication range' is left to the specific conceptual framework instantiation. In other words, the conceptual framework does not prescribe any specific group formation policy or protocol. For example, an instantiation of the conceptual framework

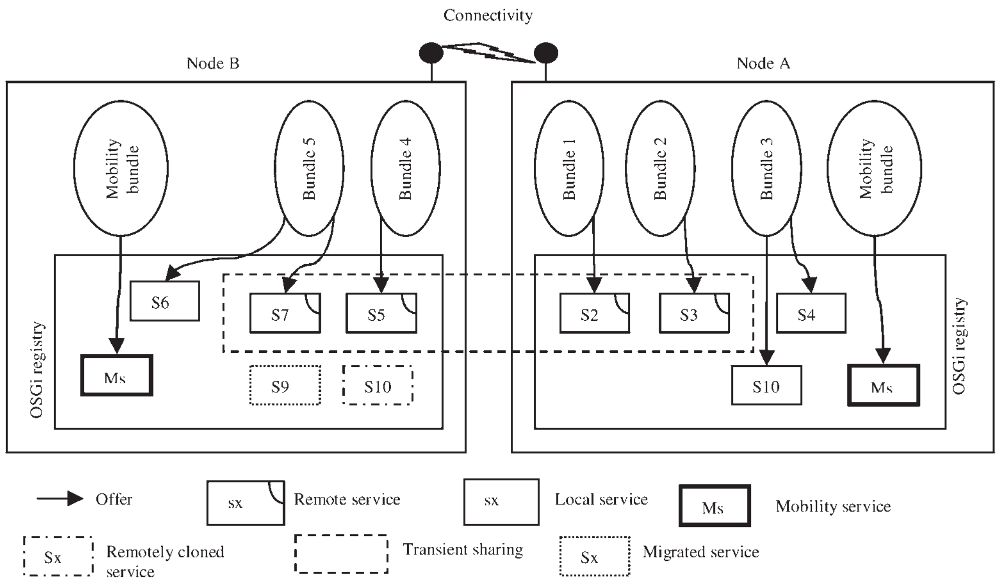

FIGURE 1. Conceptual OSGi framework.

may use the global positioning system to manage connectivity between physical nodes, where connectivity may be specified to correspond to a certain *distance* or distance range within which communication is possible or is permitted. A different conceptual framework instantiation may use a multicasting protocol to recognize as a group all nodes that have subscribed to the corresponding multicast address. Various discovery protocols such as simple service discovery protocol, Jini and the service location protocol (SLP) can all be used to detect and establish a node's connectivity status, where both wired and wireless connectivity are supported. Furthermore, the conceptual framework does not prescribe any specific communication model such as RPCs or event-based communication. The conceptual framework is also independent of any specific communication protocol. Different instantiations of the conceptual framework may use different communication protocols based on network setup and communication requirements. For example, an instantiation may use simple object access protocal (SOAP) over HTTP to exchange information between nodes, while a different instantiation may use binary messages over TCP or UDP. The conceptual framework is therefore independent of the underlying transport layer. These properties of the conceptual framework make it suitable for a variety of application scenarios and network setups such as fixed and ad-hoc networks.

As new nodes and services join and leave, the federated logical registry is *automatically* adjusted to reflect the changes. A service joins the federated shared registry when its parent bundle is started in its local node and leaves when the parent bundle is stopped. Only services that have been explicitly marked for remote access will be considered for transient sharing. In addition, a remote service can only join if its parent node has joined. A node joins when it becomes connected and leaves when it is disconnected. Joining of a node implies the joining of all remote services within that node. Conversely, a node leaving implies the disjoining of all previously joined services that reside in the departing node.

3.4. Controlled visibility and service identification

A ubiquitous environment will typically comprise many devices. The proposed conceptual framework may therefore result in a large number of services that are made visible to individual nodes. In this case, service programming may become more complex because developers are required to sift through and interact with a large number of services. In addition, certain devices may have certain resource limitations and should not be overwhelmed with services. Furthermore, some nodes may only be interested in using a specific type of services, while other nodes may only be interested in using services that are provided by specific nodes. Therefore, conceptual framework instantiations should support controlled visibility at two levels. First, at the node level, it should be possible to control visibility through the ability to hide certain nodes from certain other nodes. Second, at the service level, it should also be possible to hide certain services from certain other services. If a node is invisible, then all services originating from that node are also invisible. Conversely, if a service is visible, then its source node must also be visible.

There is also a need for a global naming scheme that uniquely identifies mobile entities across the federated nodes. The OSGi specification already identifies services within a single OSGi framework instance through a unique `Service.ID` property which is automatically assigned by the framework implementation to every newly registered service. Bundles are also identifiable in the scope of a single framework instance through a unique `Bundle-SymbolicName` manifest header. Instantiations of the conceptual framework may extend these identification schemes to cover the federated nodes. Alternatively, instantiations may provide their own naming schemes, for example, using randomly generated identifiers.

3.5. Mobility bundle

A mobility bundle implements the conceptual framework. It uses the communication infrastructure to transparently manage the transient sharing of services across nodes. A mobility bundle also offers a mobility service that provides an API for bundle and service mobility supporting both migration and remote cloning.

3.6. Mobility paradigms

3.6.1. Service distribution

The mobility bundle monitors the local OSGi registry. Bundles wishing to offer services that are enabled for remote access should explicitly mark them for remote access. The mechanism for marking these potentially remote services is implementation dependent; for example, by setting a specific REMOTE_CANDIDATE property of the service in question or by implementing and registering it under a `java.rmi.Remote` interface. The mobility bundle automatically and transparently shares these remote services so that they become accessible to all OSGi nodes that are connected. For example, in Fig. 1, Bundle2 is able to use a remote service 'S7' that is offered by Bundle5 as if Bundle5 was residing in the same Node A. However, the mobility bundle should not completely hide from clients the distributed nature of remote services because certain considerations such as performance and security may be important for the client bundles and services (see Section 4.3). For example, Bundle2 in the figure should be aware of the distributed nature of service S7 so that Bundle2 can decide whether and how to use the service S7.

3.6.2. Other mobility paradigms

It is also possible to instruct the mobility bundle to migrate a specific service to a specific remote node. In this case, the service is first withdrawn from the source node before re-offering it in the destination node. Remote cloning is also supported, where a copy of the service is offered in the specified destination node. Mobile entities can only be moved/cloned between *connected* nodes. Migration and remote cloning work in both directions where a bundle or service can either be *sent* or *retrieved* from a remote node. The service 'S9'' in Fig. 1 is an example of service migration from 'Node A' to 'Node B', whereas service 'S10' represents a remotely cloned service.

3.7. Synchronous versus asynchronous mobility and invocations

Some communication infrastructures such as publish-subscribe are *asynchronous* by nature, which means that support for mobility on top of them is also asynchronous, i.e. an initiator of service migration will not block waiting for the service to reach destination. In certain scenarios, support for *synchronous* mobility on top of an inherently asynchronous communication infrastructure may also be required. It is possible to use the *Half-Sync/Half-Async* design pattern [36] to efficiently integrate both the synchronous and asynchronous models. This pattern allows for high-level tasks to use a synchronous communication model on top an asynchronous model wherein communication between entities in the synchronous and asynchronous layers is mediated through an intermediary queue. Such use of the Half-Sync/Half-Async pattern facilitates the implementation of synchronous mobility operations such as synchronous migration and cloning on top of the inherently asynchronous communication infrastructure.

Alternatively, the Future idiom [37] can be used to provide a similar support for synchronous interaction on top of an asynchronous communication model. A future is a *place holder* for a result that will *eventually* become available after completing the asynchronous operation. When a method is invoked, it *immediately* returns a place holder, i.e. a *future object* instead of the real result. The underlying asynchronous communication infrastructure is then used to satisfy the call. Eventually when a result is returned, it is put inside the future. The client can issue the *synchronous* call, get the future object and do something else until the real result is required at which point it tries to obtain it from the returned future. If the asynchronous computation had ended by then, a result would be available from the future. Otherwise, the client will be blocked by the future until the result is available. Walker *et al.* [38] describe an RPC execution facility that adopts the future pattern and adds support for managing replies.

The proposed conceptual framework does not prescribe any specific synchronization model with respect to bundle and service mobility. Instantiations can support either synchronous or asynchronous mobility according to application requirements. In synchronous mobility, an initiator of service migration typically blocks waiting for the mobile entity to reach its destination. Typical instantiations should support both synchronization models in order to be useful for the widest possible range of applications.

4. A SERVICE DISTRIBUTION AND MOBILITY ARCHITECTURE

An architecture for the proposed conceptual framework is depicted in Fig. 2 where a mobility extension is responsible for exporting OSGi services that are marked for remote access in the source node and their subsequent import at the destination node. In addition, a mobility extension provides an API that can be utilized by applications to send and receive OSGi bundles and services between OSGi nodes.

The proposed architecture distinguishes between service discovery and interaction protocols. For instance, Jini and SLP[8] are examples of discovery protocols, whereas Java RMI [39, 40] and SOAP[9] are examples of interaction protocols. A client wishing to use a remote service must discover the exported service using the relevant discovery protocol and interact with the discovered service using the relevant interaction protocol. Conceptual framework instantiations may support different protocols for discovery and interaction. A mobility extension design may utilize the Whiteboard [41] and Factory [42] design patterns to dynamically select appropriate exporters/importers. In such design, factories are registered with the local OSGi registry and dynamically discovered by the mobility extension to obtain a suitable exporter/importer. Bridges between discovery protocols where a service is discovered using a different protocol from the one used to publish the service as well as interprotocol collaboration could potentially be supported (see for example, [43, 44] and the Device Access Specification's Driver Bundles [12]) but we do not discuss them further.

The details of the export and import mechanisms are dependent on the specific conceptual framework instantiation but follows a common approach which can be summarized as follows. Each marked service is exported in a protocol specific way. For example, a service could be exported by making it discoverable through SLP or Jini and accessible through either the SOAP protocol or Java RMI. In the destination node, such exported services are discovered using the relevant discovery protocol and a stub or proxy is automatically generated for accessing the remote service using the relevant interaction protocol (export-side proxy generation may also be supported, see Section 4.4.3). Messages are exchanged between nodes to convey service availability events, invoke remote operations or to request migration of an OSGi entity, as few examples.

Depending on the specific conceptual framework instantiation, subcomponents of the mobility extension may be implemented as separate bundles. A mobility extension is

[8]http://tools.ietf.org/html/rfc2608.
[9]http://www.w3.org/TR/soap/.

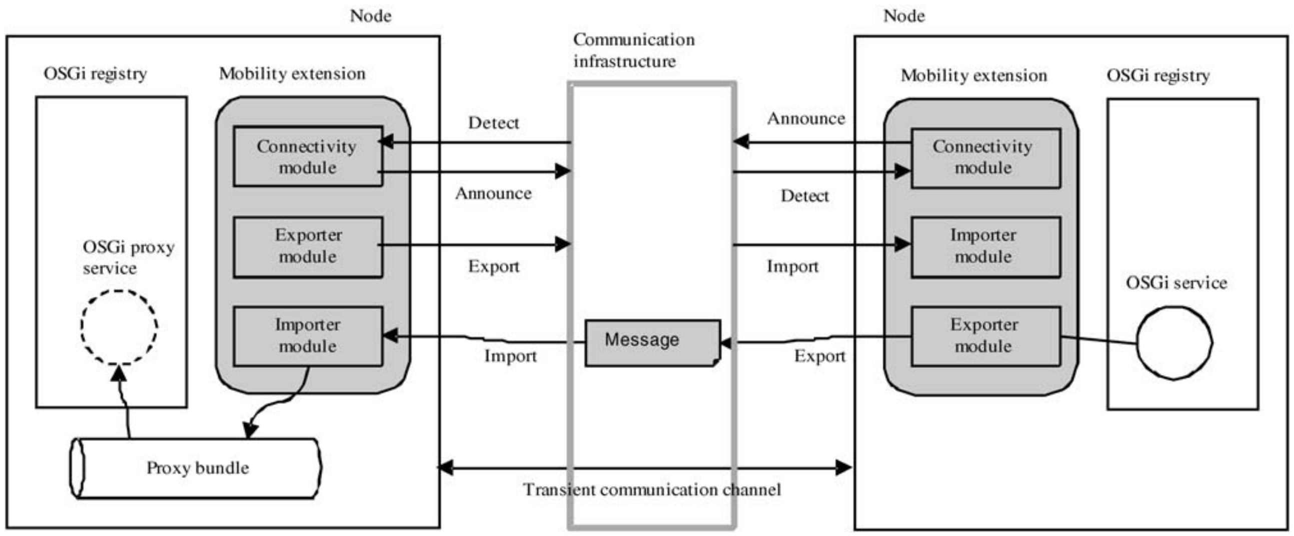

FIGURE 2. OSGi mobility and distribution architecture.

responsible for managing the connectivity between nodes and the transient sharing of exported structures. It is also responsible for communicating exchanged messages between connected nodes and for establishing and managing a communication channel between the connected nodes. Trust and security aspects including privacy of communication, for example, through encryption techniques, can be supported but no assumptions are made about the supported trust and security aspects.

The communication infrastructure between nodes may be configured differently depending on the application requirements and the support provided by the underlying communication infrastructure implementation. For example, in a centralized configuration, a single communication server is responsible for routing messages between connected nodes. In a distributed configuration, nodes connect to multiple access points that are connected to each other forming a coherent distributed system.

4.1. Deployment bundles

The OSGi platform uses a bundle as the unit of delivery and deployment. Consequently, mobile bundles can be readily deployed on arrival at the destination node. Mobile services, however, must be packaged as bundles before they can be deployed in the destination node. When transferring individual services, the mobility extension automatically packages them into bundles for subsequent deployment. This packaging of mobile services into deployment bundles can happen either at the source node or on arrival at the destination node.

4.2. Push and pull mobility styles

In a push style of mobility, e.g. REV, an initiator of migration or cloning resides in the source node. The initiator can be either the mobile entity itself or a third party. On initiation of a move, the mobile entity is first uninstalled (in the case of migration) and a corresponding message is constructed and published via the communication infrastructure. The destination node detects and interprets the message and then installs and starts the corresponding deployment bundle which offers a *new* service as opposed to a *proxy* service.

In a pull style of mobility, the initiator resides in the receiving node. The initiator uses the mobility extension to request the migration or cloning of a specific service from a remote node. In response, the mobility extension publishes a corresponding request via the communication infrastructure. The mobility extension in the source node responds by pushing the required entity using the communication infrastructure. When received in the destination node, the response message is used to install and start a corresponding deployment bundle as described above.

4.3. A note on implementing the proposed conceptual framework

At an implementation level, there are essentially two mobility mechanisms that have been adopted by MA systems: *middleware technologies* such as Java RMI and *network sockets* [30]. The relationship between MA systems and middleware technologies such as CORBA, DCOM and Java RMI is that such middleware technologies offer low-level primitives for mobility that are used by MA systems [32]. Java RMI is the most widely used mobility mechanism especially for Java-based agent systems. Using network sockets, a source node directly serializes the agent and its resources which are then communicated through sockets to the destination node. Although the use of Java RMI for inter-node communication has the advantage that it supports automatic class loading, a number of authors have argued against using Java RMI for MA migration. For example, according to [45, 46], Java RMI does not provide the programmer with control over the strategy for class relocation. There is also no way to relocate a set of needed classes in a *single* operation which counters the benefit of MAs in terms of reducing network overhead and communication costs. Specifically, 'the impossibility to specify dynamically the constraints on relocation of classes and to limit dynamic linking hampers a full exploitation of MA paradigm [45]'. Furthermore, RPC distribution technologies such as Java RMI are often synchronous in nature where both client and server need to be available at the time of communication. The characteristics of ubiquitous environments in terms of device mobility, variable network topology and frequent unannounced disconnections mean that an asynchronous interaction style may be more desirable for agent migration and internode communication. For example, a client may connect to a service provider, make a request, disconnect (either voluntarily or involuntarily) and later reconnect to collect the results. Asynchronous interaction styles are considered as a more appropriate communication model particularly for mobile environments (see for example, [20, 47–50]). Our conceptual framework based on the GVDS concept supports certain levels of decoupling in both time and space where nodes can exchange information indirectly through the shared structure (or the asynchronous communication infrastructure if one is used).

Many distributed systems attempt to unify the object model by making distributed computing more like local computing with the aim of simplifying distributed computing. Kendall *et al.* [51] discuss the nature of distributed computing and the importance of recognizing the fundamental differences between distributed and local computing. They argue that developing a distributed system requires awareness of *latency* and the different model of *memory access*. In addition, it requires the programmer to explicitly consider the issues of *concurrency* and *partial failures*. The same authors argue that unifying the object model by designing all interfaces as if they were remote or by designing all interfaces as if they were local are both flawed and counter-productive. Ignoring the differences between distributed and local computing can and will cause performance, robustness and reliability problems, and therefore distributed systems that attempt to paper over these differences are deemed to failure. These highlighted issues were the motivation behind the development of Java RMI as a technology for distributed computing. The same authors suggest that these highlighted issues are more significant in p2p-distributed object-oriented applications as the case in the target domain of this paper. However, Rellermeyer *et al.* [52] and despite their agreement with the above arguments argue that the OSGi application model is fundamentally different from the pure object-oriented application types originally targeted by [51]. These differences largely stem from the characteristics of service-orientation, specifically: (1) location independence; (2) substitutability; (3) higher granularity. Rellermeyer *et al.* [52] propose a simple and pragmatic alternative OSGi-specific solution which is to map network failures and (latency, partial failures and concurrency non-determinism) to local unplug events by withdrawing the corresponding proxy service in the client node. The assumption is that all OSGi applications are written with no guarantee as to the continuous availability of used services (which follows from the nature of the OSGi application model itself) and that guards are put in code to detect and react to such service departure events. Hence, by mapping network failures to service departure events that are already handled by client applications, clients will simply react appropriately, e.g. by searching and binding to a replacement service. Note that remote servers could still raise non-network exceptions and these must not be hidden from the local clients. We in turn agree with those arguments by [52] but add that service location, cost, reliability and other similar information can be very important for certain clients, for example, clients wishing to use specific resources which can not be easily relocated, e.g. hardware resources. Another example is sensitive and confidential information which should not be exposed to or processed by non-local services. Therefore, we do not desire absolute transparency. To address these concerns, location, cost and other similar information can be attached to proxy services in the form of service properties for inspection by clients during service selection and invocation. Implementations of our proposed conceptual framework should consider the above issues when distributing OSGi services through transient sharing.

The above discussion only concerns client–server interactions when sharing the OSGi services. Other mobility paradigms are unaffected because the programmer is explicitly aware of the location where computation takes place. To summarize, our proposed conceptual framework does not preclude the use of traditional RPC technologies such Java RMI for communication between connected nodes. Asynchronous communication infrastructures can also be used to provide decoupling in time and space. In particular, the provisioning of location aware primitives to GVDS may subsume the traditional client–server paradigm as a special case of GVDS [34]. In the following section, we describe a mobility mechanism that is the result of adapting the μCode mobility toolkit [45] for OSGi bundle and service mobility.

4.4. Briefcases for service and bundle mobility

In addition to its code, a mobile entity may also be associated with a set of resources that may need to be transferred along with the entity. The purpose of a briefcase is to act as a container that can be filled arbitrary with entities and resources (or information about where they can be found) for shipment to the remote node as a *single unit*. The concept of a briefcase was first proposed in [53] and is also adopted by [45] who call it a *group*. A briefcase provides an API for populating and inspecting the briefcase's content. The proposed architecture considers a briefcase as the minimum unit of mobility that is communicated to peer nodes as part of the exchanged message. The benefits of using the briefcase abstraction can be summarized as follows:

(i) It provides the developer with a single API for bundle and service mobility irrespective of the chosen mobility paradigm.
(ii) A briefcase is the container for the transferred code and resources. Depending on application requirements, it is possible to create different briefcase implementations with different capabilities. For example, although the granularity of the briefcase content is left to the application developer, a briefcase implementation may enforce the semantics that a briefcase can only be used to transfer a *single* bundle or service at a time along with their used resources. Other briefcase implementations may exclusively support specific resource relocation strategies such as rebinding or remote references.
(iii) The concept of a briefcase and its API simplifies the process of finding the relevant resources when the mobile entities arrive in the destination node which in turn facilitates the generation of the corresponding deployment bundles (see Sections 4.1 and 4.4.3).
(iv) Additional features could also be supported by briefcase implementations. For example, a briefcase could be made responsible for automatically calculating the transferable code as classes and resources are added to the briefcase.

When a bundle or service is to be migrated or transferred for REV or remote cloning, its byte code is packaged into a briefcase that is sent over the network to the remote node. At the destination node, the byte code is processed and loaded into the running system. The next section describes how the transferred byte code is calculated in the source node, while Section 4.4.2 describes how it is processed and loaded in the destination node. These solutions to calculating the transferred code and its subsequent integration in the receiving node are based on approaches described in [45] and [54], respectively.

4.4.1. Automatic calculation of the transferred code

If the developer wishes to transfer a whole bundle, the bundle's byte code is read and packaged into a briefcase in accordance with the chosen mobility paradigm, e.g. cloning or migrations. If the developer wishes to transfer only a specified service, then the transferable code is automatically calculated by analysing the service interface(s). For each implemented interface, the analysis mechanism automatically determines all the classes that must be transferred in order to offer the service with the remote node under the given interface. This automatic calculation of the transferred code is very powerful and greatly simplifies application development.

In our current prototype implementations, calculating the transferable code is the responsibility of the briefcase which has some intelligence built-in. For example, when a class is added to the briefcase, the briefcase automatically computes and adds the full closure of the class. To prevent certain classes or whole packages from being automatically added, they can be programmatically specified as *ubiquitous*. Ubiquitous classes and packages are assumed to be available in every node and hence will not be transferred. By default, system classes (`java.*` and their subpackages), OSGi runtime (`org.osgi.*` and their subpackages), and the mobility extension packages (which may vary depending on the specific implementation) are all considered ubiquitous. The list of ubiquitous packages contributes to the *Import-Package* header of the deployment bundle at the destination node.

The current briefcase implementation uses the following algorithm for calculating the transferred code:

- Include all interfaces under which the service was registered with the OSGi registry and their full class closures, i.e. recursively include all reachable classes and interfaces, subject to the ubiquity specifications (see below).
- For each explicitly added class (explicit injections), include both the class and its full class closure.
- Interfaces and classes defined by packages `java.*`, `org.osgi.*`, and the mobility extension packages are not included—they are assumed to be available at the destination node(s).
- Interfaces and classes that are explicitly specified as ubiquitous are not included. Our prototype implementations provide several methods for specifying class ubiquity at different levels of granularity. For example, specifying a package as ubiquitous automatically considers all classes within that package as ubiquitous. If the package name ends with '.*', then all the subpackages are made ubiquitous too. Alternatively, ubiquity can also be specified on class by class basis.
- For explicitly added serializable objects, their whole object graph is serialized automatically. In order to reconstruct these objects at the destination node, the involved classes are automatically included, subject to the ubiquity specifications.
- Our prototypes also automatically calculate the deployment bundle's Import-Package header based on the above steps.

4.4.2. Integration of the transferred code in the receiving node

When a whole bundle is transferred, it is simply installed and started on arrival in the destination node. For service mobility, the mobility extension automatically generates, in the destination node, a *deployment bundle* that encapsulates the transferred service. A deployment bundle is needed in order to integrate the transferred service with the local OSGi framework in the destination node (see below). The mobility extension dynamically generates, installs and starts this deployment bundle which then offers the encapsulated transferred service.

Deployment bundles are required for integrating the transferred code in the destination node. When a service is transferred, the mobility extension includes all interfaces and classes that are required in order to offer the service in the destination node. A remote service candidate may be using interfaces and classes from different bundles (imported packages). Such imported packages are considered as environment dependencies and hence must not be transferred, which is the default briefcase behaviour. At the destination node, these package dependencies are simply reimported, i.e. resource binding by type. Different briefcase implementations may change this default behaviour, e.g. by enforcing dynamic remote linking policy. The mobility extension must ensure that these package and class dependencies are satisfied at the destination node. The OSGi specification defines a dependency resolving process which ensures that package and other environment dependencies are satisfied before a bundle can be allowed to start. This dependency resolving is automatically performed by the OSGi framework after a bundle is installed but before it is started. This means that at the destination node, there is a need to integrate the transferred service at the *module layer* [54]. The only mechanism available in the OSGi platform for triggering this resolving process is by installing and starting a bundle which causes the resolving of that bundle. Therefore, the deployment bundle approach seems to be the only way currently available for achieving such integration at the OSGi module layer.

4.4.3. Generating the deployment bundle in the source node

In certain scenarios it may be more efficient to generate the proxy bundle on the exporting source node as opposed to the destination, for example, in order to preserve resources in importing devices with limited resources. The current prototype implementations can be customized to generate deployment bundles in either the destination or source nodes. In the latter case, a deployment bundle is pre-generated in the *source node* before it is shipped to remote nodes where it can be directly installed and started.

4.5. Supporting controlled visibility and uniform resource identification

As discussed in Section 3.4 there is a need to support controlled visibility. At the node level, different instantiations of the conceptual framework can define different rules for connection and disconnection which provides them with control over node visibility. In addition, our current mobility extension prototype supports the two concepts of *import filters* and *views* which are described below.

An import filter is any object of type `IImportFilter` that implements a single method **public boolean** accept(String serviceURI, Dictionary properties). When associated with the mobility extension implementation, the filter is called whenever a remote service is about to be imported. Only if the accept method returns true then the service is imported. This is a straightforward use of the strategy design pattern [42] to encapsulate the filtering logic. The current prototypes come with three default filter implementations:

- *UniversalImportFilter*. This is the filter that is used by default which imports all available services.
- *ImportByHostFilter*. This is a filter that selectively imports bundles and services from certain nodes which are specified when the filter is instantiated.
- *ImportByServiceTypeFilter*. This is a filter which filters services based on their service types.

It should be possible to build on import filters to support the notion of an *acquaintance* list that represents a set of neighbouring nodes within communication range which this node considers interesting. It is expected that developers will provide their own filter implementations based on application requirements. For example, developers can create filter implementations that filter out services based on specific service properties in which they are interested.

Service-oriented systems (SOSs) are usually composed of several inter-dependent services. Service providers may choose to offer or withdraw their services at any time whereas clients are free to use any available service at runtime (substitutability). Accordingly, SOS need to compose and adapt to services dynamically [11]. However, programming service dependency management (SDM) especially in ubiquitous contexts is complex and error prone task because [8]: (1) Developers may have to deal with large number of communication protocols (2) Typical current middleware does not provide adequate support for SDM.

A number of OSGi SDM mechanisms have been proposed in the literature (see for example, the Service Tracker [12] , Xenotron [55], Declarative Services [12], iPOJO[10] and Service-Binder [23]) that aim to simplify OSGi application development through support for constant monitoring and measures for adaptation. Essentially, an SDM mechanism mediates between a component and the services registry to (1) manage the component's dependencies on other services available in the registry; (2) manage the component's provided services by automatically handling their registration in the registry. The specification of provided services and required dependencies is implementation dependent and can be defined either programmatically or using XML description files. The SDM mechanism is responsible for tracking required

[10]http://cwiki.apache.org/FELIX/ipojo.html.

dependencies which are then *injected* into their respective dependent services. If instructed, the SDM mechanism can also handle the registration of provided services with the OSGi registry on behalf of the component. The SDM mechanism can manage the life cycle of the provided service by ensuring that it is activated only when its required dependencies are satisfied. An active service means that: (1) all its required dependencies are satisfied according to the dependencies specification; (2) the service has been registered with the services registry and can therefore be used by other clients and services.

To support controlled visibility at the service level, we have created an innovative service dependency manager with support for views as in database systems. This dependency manager supports the definition of a view as a projection over the OSGi registry which is then associated with specific dependent services. Once associated with a service, that service can only track and use other services that are visible through the view. Views can be merged, intersected, and differed as in set theory. In addition, they can be changed dynamically for any given service and the dependency manager is responsible for automatically adapting to changes in the new view. Views provide basic support for services scoping and grouping which is useful for predictable service composition.

Our architecture also supports a simple resource identification scheme. Every node is associated with a name consisting of the tuple `('urn:d-osgi', hostIPAddress, dosgiPortNumber)` which uniquely identifies a node in the federation, i.e. it describes a placeID. Subsequently, every service can be uniquely identified across the federation through the tuple `(placeID, Service.ID)`. However, this approach has the limitation that the `Service.ID` property is *not* persistent across OSGi framework restarts. Consequently, stateful service interactions across OSGi framework restarts cannot be supported. It is possible to overcome this limitation by using the standard OSGi `Service.PID` property which describes a persistent service identifier across OSGi framework restarts. However, unlike the `Service.ID` property which is automatically generated and assigned by the OSGi framework implementation to every registered service, the `Service.PID` property must be generated and assigned by the application developer at bundle deployment time.

5. DESIGN AND IMPLEMENTATION OF THE D-OSGI MOBILITY EXTENSION

This section describes the design and implementation of the *D-OSGi bundle* which realizes the proposed conceptual framework and architecture using the Lime middleware [56] as an underlying GVDS implementation. We used version 2 of Lime (called LimeII [57]), which is a complete re-engineering of the original Lime implementation. LimeII differs from the original Lime implementation in various aspects that can be categorized as: (1) Improved support for mobile ad-hoc environments; (2) Support for unannounced disconnections; (3) Improved reliability and fault-tolerance; and (4) Design improvements in terms of flexibility and modularity. Our D-OSGi extension realizes the proposed conceptual framework supporting the four design paradigms client–server, COD, REV and weak MA. The section also describes variations of the D-OSGi implementation that demonstrate the flexibility and portability of the proposed conceptual framework and architecture.

The Lime middleware adapts the tuple space concept popularized by the Linda coordination middleware [58] to mobile environments by partitioning the shared tuple space into a number of tuple spaces. Each of these tuple spaces is called an interface tuple space (ITS). Each ITS is then *permanently* and *exclusively* associated with a single process. An ITS represents the single point of interaction for the owning process. A process uses normal tuple space operations to add and retrieve tuples to/from its ITS. The data in an ITS is the only data available to the owning process while the process is alone. When two or more processes (or hosts) are connected, the content of all the individual ITSs are shared to form a single shared tuple space. Our D-OSGi implementation adapts Lime to OSGi service distribution by associating a single ITS with each node which is then transiently shared when the nodes are connected.

5.1. D-OSGi components

The main components of the Lime-based D-OSGi mobility extension are shown in Fig. 3. It adopts a modular design consisting of four main stationary software agents with well defined roles, which are described below.

5.1.1. *Exporting agent*

This is a singleton (per D-OSGi bundle) agent that is responsible for distributing remote service candidates to reachable nodes according to the client–server paradigm. An exporting agent monitors the local OSGi registry for remote service candidates. When the agent detects the registration of a remote service candidate, it automatically maps the service into an `ExportTuple` that is inserted in the ITS. Similarly, when the agent detects deregistration of a remote service candidate, it automatically propagates this information to reachable nodes by inserting a `ServiceWithdrawnTuple` which will be detected by other importing nodes and the appropriate action will be taken. In addition, the Exporting Agent detects property updates of remote service candidates and automatically inserts corresponding `PropertiesUpdateTuple(s)` in the ITS in order to propagate the updates to reachable nodes. Another responsibility of this agent is to launch and maintain a response sender request receiver agent for every exported service (see Section 5.1.4 for details).

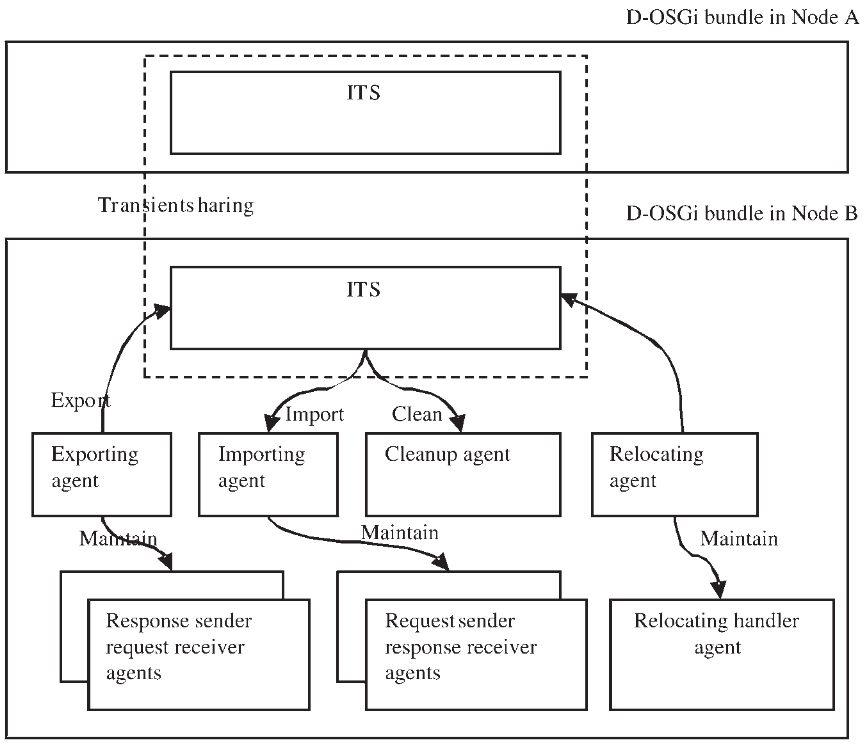

FIGURE 3. D-OSGi components.

5.1.2. Importing agent

This singleton agent (per D-OSGi bundle) continuously monitors the ITS for `ExportTuple`(s) and maps them to proxy services which are subsequently registered with the local OSGi registry (subject to import filter specifications). Another responsibility of the Importing Agent is to monitor the ITS for `PropertiesUpdateTuple(s)` and update the properties of the corresponding proxy services. It also detects the departure of exporting nodes and unimports all services previously imported from those departing nodes. Finally, it detects when previously imported services are withdrawn, by monitoring the ITS for `ServiceWithdrawnTuple(s)`, and automatically unimports those withdrawn services. A service is unimported by stopping and uninstalling its corresponding proxy bundle. Client bundles using the withdrawn proxy service will then be notified by the OSGi eventing mechanism and, based on the OSGi application model, should react accordingly, e.g. by searching and binding to alternative services. If a remote client D-OSGi bundle does not unimport a withdrawn service, it is considered as a programming error. All calls on such unreleased but withdrawn services (i.e. stale references) will fail returning a `ServiceUnavailableException`. The client D-OSGi bundle may catch such exceptions and force un-importing of the corresponding service.

In comparison, garbage collection in Jini relies on leases to grant access to offered objects on a time basis. It is the client's responsibility to renew the lease before the current one expires otherwise the offered object is discarded by the lookup service. A client is not expected to use an object whose lease has expired. The need to frequently renew leases may place additional load or burden on clients, e.g. when the cost of connection is high. Jini introduces the Lease Renewal Service specification to try overcome this limitation.

5.1.3. Request sender response receiver agent

This is a client side agent that is responsible for forwarding invocation requests on the proxy services to the remote node hosting the original service and for receiving the responses. A different instance of this agent is assigned to each proxy service in the importing node. The justification for having multiple instances of this agent is that method invocation is synchronous where a client must wait for the result to be returned. If there was a single agent instance for handling all the proxy services, then all invocations on any of those proxy services will have to compete with each other for the agent's attention where the agent has to forward each invocation and wait for its reply before handling another invocation. This is clearly unacceptable because it can greatly degrade performance. By assigning each proxy service a separate agent instance, each proxy service effectively will have its own thread of control for forwarding invocation requests and receiving replies.

A proxy service may be invoked concurrently by multiple clients. In this case, if one of the invoked methods blocks or slows considerably in returning a result, the request sender response receiver agent for that proxy service must still be able to handle the other concurrent method invocations on the proxy service while waiting for the current slow invocation to return. The current D-OSGi implementation uses the active object design pattern [59] to meet this requirement and to decouple method invocations from their subsequent execution by the request sender response receiver agent. The decoupling is achieved by transparently and automatically converting method invocations into method request objects that are then passed to a different thread of control. The implementation uses a queue for storing these method request objects. A scheduler continuously runs in the other thread of control to dequeue and process method request objects.

5.1.4. *Response sender request receiver agent*

This is a server side agent that is responsible for receiving service invocation requests from client nodes and sending back the responses. A different instance of this agent is assigned to each exported service in the offering node. The justification for having multiple instances of this agent is similar to that of request sender response receiver Agent; if there was a single agent instance for handling all the exported services, then all received invocations on any of those exported services will have to compete with each other for the agent's attention. In such case, the agent has to invoke the received invocation request on the exported service and send back the reply before handling another received request which is clearly unacceptable because it can greatly degrade performance. By assigning each exported service a separate agent instance, each exported service effectively will have its own thread of control for receiving requests and sending replies. Similar to the request sender response receiver agent, an active object design pattern is used to decouple method invocations from their execution.

Use of the request sender response receiver and response sender request receiver agents supports concurrency at two levels. First, at the service level, each service is given its own thread of control. Second, at methods level, invocations are decoupled so that they do not get in each other's way. Such decoupling enables the D-OSGi implementation to efficiently scale up as the number of services and methods increase. It also improves the end-to-end quality of service for both clients and servers. Furthermore, the two agents encapsulate the RPC mechanism and its implementation (see Section 5.2).

5.1.5. *Relocating and relocating handler agents*

The relocating agent is responsible for service mobility in terms of migration and remote cloning. Both push and pull style mobility are supported. The relocating handler agent in the destination node responds by creating new services that are installed and started locally. These created services are different from the proxy services created by the Importing Agent in that they do not forward method calls to the remote providing node but rather execute the calls locally. The relocating agent and the relocating handler agent are essentially a design and implementation choice. For example, it is possible to merge them with the exporting and importing agents respectively.

5.1.6. *Clean-up agent*

This *optional* singleton agent (per D-OSGi bundle) is responsible for cleaning up expired tuples that were inserted by the parent node. For example, when the properties of an exported service are updated, a corresponding `PropertiesUpdateTuple` is inserted in the ITS. This inserted `PropertiesUpdateTuple` will be detected by importing nodes which will subsequently update their corresponding proxy services. After all importing nodes had updated their corresponding proxy services, the tuple is no longer needed. New property updates will result in new `PropertiesUpdateTuple(s)` being inserted in the ITS. `ServiceWithdrawnTuple(s)` are another example of tuples that should be eventually cleaned. For performance and efficiency reasons and in order to prevent the tuple space from growing very large, this agent automatically cleans up those expired tuples. The need to insert new `PropertiesUpdateTuple(s)` and for `ServiceWithdrawnTuple(s)` is a result of the limitation of Lime's *reaction* semantics [56], which we use to monitor the ITS. Lime's reactions can only fire in response to the *insertion* of new tuples as opposed to the *withdrawal* or *modification* of existing tuples. Enhancing the semantics of Lime's reactions will allow us to *reuse* the same tuple to propagate new information and will obviate the need for the clean-up agent. Alternatively, some tuple space implementations readily support tuples that can be associated with a time-to-live parameter which is used to automatically remove the tuple when this time period expires.

5.2. D-OSGi variations

The Lime middleware supports the extended *out*[λ] operation which takes a location parameter (λ) specifying the target destination of an inserted tuple. Location parameters make it possible to exclusively communicate migration and clone messages to specific remote nodes. Further description of the semantics of the *out*[λ] operation is given in [56]. In a first D-OSGi prototype (Prototype A), we successfully used the underlying tuple space and the *out*[λ] operation to forward method invocation requests and responses between the proxy services and the original services in the source node. This exercise backed the claim by [34] who suggested that the provisioning of location aware primitives to GVDS may subsume the traditional client–server paradigm as a special case of GVDS. Despite use of location parameters, exchanged messages are still shared with all connected nodes which raises security and privacy issues. To overcome this limitation, the concept of a communication session [25] could be supported. Although not currently implemented in our prototype, a session could be supported via a temporary and uniquely identified ITS that is used to securely exchange request and response tuples. Communicating nodes may exchange a potentially encrypted session ID which, after decryption, can be used to create the uniquely named ITS. Tuples exchanged through these uniquely named ITSs will only be visible to those nodes with access to the session ID and the decryption key. In addition, MAs interacting with agents from remote nodes may continue the same conversation by simply carrying along the unique session name/ID and using it to create a new ITS at the new destination node. Alternatively, the session may be terminated implicitly when one of the participating agents moves.

We also created a second prototype (Prototype B) that uses direct socket-based communication for remote invocations. A transient TCP channel (whilst the two nodes are connected) is automatically created, maintained and managed by D-OSGi. The flexibility of the D-OSGi design meant that changes were largely confined to the request sender response receiver and the response sender request receiver agents. Similarly, migration and remote cloning can be supported via either the tuple space or socket-based communication and changes will be confined to the Relocating and Relocating Handler agents.

In a third prototype (Prototype C), we used Java RMI to implement remote invocations between client and server nodes. In the source node, D-OSGi detects all services that are registered under the `java.rmi.Remote` interface and automatically exports them via Java RMI registry. The unique identifier of the exported service along with the location information of the Java RMI registry are then communicated via the ITS to all connected nodes. In the destination nodes, D-OSGi uses the received information to create a *finder* proxy service that is registered with the local OSGi registry. Client bundles use this finder proxy service to obtain the exported service's identifier and Java RMI registry location information which they then use to locate an Java RMI proxy (the remote reference) for invoking the original service using Java RMI mechanisms.

Use of a finder service is a compromize to integrate Java RMI and support automatic contextual management. We tried to get the D-OSGi bundle in the destination node to obtain the RMI proxy on behalf of client bundles and to register this RMI proxy as a service with the local OSGi registry for direct use by client bundles. Note that an RMI proxy obtained via `Registry.lookup(name)` method implements all the interfaces of the original service and can therefore be registered with the OSGi registry under these interfaces. However, we encountered a `ClassCastException` whenever a client bundle attempts to cast the RMI proxy service into one of the implemented interfaces. This is because on the unmarshalling side (in this case the destination node), Java RMI runtime creates a new classloader with the classloader of the calling class as the parent. Since, the OSGi platform associates different bundles with different class loaders, it is only possible to correctly cast the returned RMI proxy if the bundle performing the cast is the one who obtained the same RMI proxy via the `Registry.lookup(name)` method.

A fourth prototype (Prototype D) successfully used the Siena publish–subscribe system [60] to export/import remote services. However, ubiquitous environments are characterized by ad-hoc and frequent (dis)connections and therefore published messages representing service (un)availability events will not be seen by nodes that were disconnected at the time of publication. In order to support such scenarios, the used publish–subscribe implementation should also support message buffering where published messages are buffered for subsequent delivery to connecting nodes. A reconciliation functionality may also be needed to prevent duplicate messages and to guarantee message delivery and order. Many publish-subscribe implementations have been adapted for mobile environments through message buffers that hold published messages on a subscriber's behalf while it is disconnected (see for example, [47, 61–63]). Other typical extensions to the publish–subscribe interaction model include message sequence detection, queuing, and message persistence. Eugster *et al.* [20] suggest that 'message passing, remote invocations, notifications, shared spaces and message queuing do all constitute alternative communication paradigms to the publish–subscribe scheme'. The same authors describe the commonalities between these alternative communication models and the publish–subscribe interaction style. The results of this comparison emphasize the inability of these alternative communication models to *fully decouple* the interaction between the participants. However, some *modern* implementations of these alternative communication models, e.g. Lime, do support the full decoupling along the three dimensions of time, space and synchronization. Johanson and Fox [64] discuss the nature of the tuple space model and argue its suitability for communication and coordination in interactive workspaces and ubiquitous environments in general. The same authors propose a set of extensions to the basic tuple space model that facilitate their use for purposes of coordination and communication in ubiquitous environments. Further discussion about the nature and benefits of the publish–subscribe interaction model as well as classification models and comparisons of publish–subscribe implementations can be found in [20, 47, 65, 66].

6. EVALUATION

6.1. Spontaneous interactions

The main benefit of Java RMI is dynamic class loading when such classes are unavailable in the target node. When marshalling a serialized object, Java RMI annotates the serialized stream with the *codebase* of the classes that are inside the stream. If the class loader is of type `URLClassloader` that contains a *global* URL, that URL is used, otherwise the stream will be annotated with the value of the system property *java.rmi.server.codebase*. The OSGi platform does not use class loaders of type `URLClassLoader` and therefore developers need to specify the codebase property. However, setting the codebase property violates the main benefit of OSGi modularity and bypasses OSGi class loading and delegation model. Other OSGi distribution solutions such as R-OSGi (see Section 7) do not support spontaneous interactions.

For client–server interactions, D-OSGi supports Scenario 2 and Scenario 3 described in Section 3.1. In D-OSGi, the serialized stream is annotated with the uniform resource identifiers (see Section 4.5) of both the invoking and target services which are then used to dynamically load required additional resources, e.g. classes, from the relevant bundle in the other node. Consequently, D-OSGi observes OSGi modularity and supports true spontaneous interaction that obviates the need for setting a codebase property.

6.2. Performance

Based on a quantitative study and analysis, Baldi and Picco [33] investigate the use of the mobile code paradigm and its effectiveness in reducing network traffic in the domain of network management. They suggest that the decision of when to use a mobile code design paradigm in place of a traditional client–server architecture is dependent on the following two aspects: (1) the model of the management functionality to be implemented and the information about the managed network; (2) a precise quantitative description of the management protocols and the mobile code system used for the implementation. Similarly, the performance of our proposed conceptual framework implementation will depend on many factors including the chosen communication model and protocol.

Wu *et al.* [14] provide a qualitative performance analysis of three architectures: client–server, P2P-SOA and P2P-SOA with use of MAs. Their analysis show that the total computation load of both the client and the service provider is the same in all paradigms although the service provider's computation load increases slightly when a P2P-SOA with MAs is adopted. According to the same authors, the main difference is in the client's computation load which is highest for client–server and least for P2P-SOA with MAs. As for network traffic, a MA paradigm helps decrease the number of sent messages by encapsulating all the messages in a single MA that is sent instead. However, the actual size of traffic actually increases because the size of the MA is larger than the combined size of individual messages. On the other hand, use of MAs shortens the time needed to maintain an open connection between the communicating nodes which could significantly decrease connection costs. Further details about the performance analysis between the different architectures is given in [14].

Hagimont and Ismail [67] describe a comparative evaluation of the client–server and MAs models. The comparison was based on Java RMI, the Aglets MA toolkit, and a third proprietary agent tool kit implemented by the same authors. They conclude that significant performance improvements can be obtained using the MA model. Another performance model is given in [68] for evaluating the cost of MA interactions where either RPC or agent migration can be used to interact with agents in different places. When applied to a typical mobile computing scenario, the proposed performance model showed that optimal performance is achieved when a mixture of RPC and agent migration is used as opposed to pure RPC or pure agent migration. Chia and kannapan [69] describe a framework that enables the analysis of alternative mobility policies based on application characteristics. Their analysis also show that pure mobile- and pure stationary-based interactions are both sub-optimal. A similar conclusion is also reached by [28] who suggest that the choice of a specific interaction paradigm must be performed on a case-by-case basis, according to the type of application.

To evaluate the performance of our D-OSGi prototype, we adapted the benchmark described in [70] which is a benchmark for comparing the performance of Java RMI implementations. For comparison, we implemented the benchmark as OSGi service that we then distributed using Java RMI, R-OSGi and D-OSGi. The experiments were carried on two machines: (1) 3.0 GHz Pentium 4 PC, hyperthreading enabled, running Windows XP and Sun's J2SE 1.5.14, with 1 GB RAM; (2) 1.66 GHz Intel Core 2 Duo T5500 laptop, running Windows XP and Sun's J2SE 1.5.14, with 1 GB RAM. We used Eclipse Equinox as the OSGi platform implementation which is installed in both machines. The two machines were isolated from the LAN but connected to each other via Ethernet. The PC played the role of the server node exporting the benchmark service while the laptop played the role of the client importing and invoking those exported services.

6.2.1. *Binding time*

Our first experiment compared the binding time of D-OSGi with that of Java RMI. For D-OSGi, we measured the *total* time from D-OSGi detecting the registration of the remote service in the OSGi registry of the source node to the time that service's corresponding proxy bundle is received/created in the destination node but excluding proxy bundle install/start time. Hence, the binding time includes the time spent analysing the service's interface and creating the proxy bundle. It also includes the time for launching and setting up the corresponding request sender response receiver and response sender request receiver agents. For this experiment, we adopted an export-side proxy generation strategy. For Java RMI, binding time is the time needed to establish

TABLE 1. Binding times.

	Binding time (μs)
D-OSGi	20 656.565 (734 000 for the very first run)
Java RMI	8590 (266 000 for the very first run)

Recorded values are averaged over 100 runs.

the connection to the source node and download the stub from the codebase.

As shown in Table 1, D-OSGi's performance is considerably slower than Java RMI. The reason for this poor performance is that the used Lime implementation makes extensive use of default Java serialization which greatly degrades performance. For example, upon a tuple read or take, Lime must find a match tuple that is then returned. Because a tuple must not be removed from the tuple space on read operations, Lime makes and returns a deep copy of the tuple in question. Deep copying in Lime is implemented by serializing the tuple and then immediately deserializing it to obtain the deep copy. Worst still, all match operations return deep copies of found tuples, which means that tuples are deep copied even for take operations, i.e. a take operation deep copies the tuple before removing the tuple and returning the deepcopy. Similarly, `ReactionEvent(s)` wrap a deep copy of the triggering tuple. We have found default Java serialization to be a major performance bottleneck. For example, in an earlier D-OSGi prototype that used sockets to communicate with peer nodes, we relied on default Java serialization to deserialize and discover the types of received message requests and replies. Although our RPC implementation used socket-based communication, performance was very poor. Through simple and straightforward serialization optimizations e.g., by encoding type information, we managed to reduce the communication overhead by a minimum of 63.76% for *ping int[25600]* and a maximum of 99.8% for *ping void* and *return int*. These results are consistent with those of [70] who suggest that Java object serialization represents at least 25% of the cost of a remote invocation. We therefore believe it is possible to greatly improve the system's performance by overcoming Lime's serialization bottleneck.

The current D-OSGi prototypes use a publish–subscribe communication model to announce service availability events to all connected nodes. The publish–subscribe interaction style is *anonymous* where publishers do not have to know the consumers, and *multicasting* in nature where it is possible to send an event to multiple consumers in a *single* `publish()` operation [20]. These two characteristics differentiate the publish–subscribe style from other interaction schemes such as: message passing and RPC [20]. Resulting benefits include potential for scalability and support for dynamic adaptation [20]. Padovitz *et al.* [71] discuss the use of publish–subscribe for communication in multiagent and MA applications. They argue for the usefulness of the publish–subscribe model for this domain and report their experience adapting an existing publish–subscribe infrastructure for MA applications.

6.2.2. *Service invocation*

Our second set of experiments measured D-OSGi's cost of service invocation in comparison with both R-OSGi and Java RMI. In the tests, the client node calls various methods of the remote service using different arguments of varying size and complexity. Each recorded time value is the average of at least 100 invocations. The round trip time between the two machines in the experiment setup is measured as less than 1 ms (using command line ping). Table 2 shows the experiment results for the case when the invocation parameters are of primitive or standard types. The results show that while R-OSGi improves on the performance of Java RMI, D-OSGi (prototype B) performs marginally better than both Java RMI and R-OSGi.

We also measured the time cost of service invocation when the invocation arguments are objects of non-standard types. We set up the experiment so that the types of these arguments are available in the destination importing node but not the exporting source node. For Java RMI, these types were made remotely accessible from the source node via a lightweight fast web server, AnalogX Simple Server[11] which we installed in the destination node. Table 2 shows the results of this experiment. Note that Prototype A implements synchronous interactions over the asynchronous tuple space using the Future idiom (see Section 3.7) and hence calculation of its invocation times is similar to the other systems—we start the timer just before the invocation and stop it when the result (actual result not the future object) is received. As shown in Table 2, The invocation cost of D-OSGi (Prototype B) is slightly higher than that of Java RMI which is a small price to pay considering the added support for spontaneous interaction and conformance to the OSGi application model. The cost of D-OSGi (Prototype A) is an order of magnitude larger than the other systems which follows from the overhead of using the underlying tuple space for internode interaction. More optimized tuple space implementations (see Section 7) promise much better performance. On the other hand, use of the tuple space paradigm provides decoupling in time, space and synchronization which is better suited to the characteristics of pervasive environments. Although the current response time of Prototype A is high in comparison with the other systems, it is still perceptually instantaneous [64]. As cited by [64], studies have shown that a user would perceive a response as instantaneous when the response time is between 10 ms to 1 s depending on the type of user action. With the exception of certain commands with very large arguments, the measured response time for Prototype A is well under the threshold of 100 ms; the point at which

[11]www.analogx.com.

TABLE 2. Remote method invocation times (μ)

Remotely Invoked Method	Java RMI	R-OSGi	D-OSGi (Prototype B)	D-OSGi (Prototype A)
Primitive and standard argument types				
ping void	964.506 ± 47.162	972.894 ± 13.128	632.469 ± 5.288	12231.429 ± 214.305
ping int	988.536 ± 9.977	770.696 ± 96.107	652.280 ± 10.435	11 953.750 ± 108.736
return int	980.625 ± 12.922	775.887 ± 51.835	658.240 ± 8.109	12 611.429 ± 161.283
ping (int,int)	948.226 ± 61.515	974.619 ± 8.999	664.197 ± 7.959	11 872.857 ± 141.997
ping byte[100]	1161.905 ± 30.537	1054.922 ± 56.141	772.644 ± 7.379	12 226.250 ± 462.167
ping byte[200]	1221.591 ± 23.646	1161.364 ± 16.389	846.024 ± 10.715	11 920.000 ± 72.899
ping byte[800]	1685.877 ± 91.161	1647.530 ± 16.352	1313.636 ± 11.929	11 962.857 ± 77.591
ping byte[102400]	114 415.000 ± 355.106	114 172.857 ± 390.374	112 988.750 ± 686.321	14 2401.250 ± 2242.663
ping int[100]	1397.500 ± 16.073	1317.308 ± 13.370	1005.128 ± 13.241	11 762.857 ± 67.763
ping int[200]	1738.245 ± 23.511	1642.045 ± ?	1315.962 ± 0.893	11 785.714 ± 78.895
ping int[25600]	113 304.286 ± 446.410	113 260.000 ± 444.136	113 928.571 ± 611.776	15 4130.000 ± 2658.136
ping float[100]	1386.796 ± 0.914	1322.051 ± 1.256	1014.382 ± 12.463	11 852.857 ± 132.634
ping float[200]	1725.531 ± 39.518	1660.227 ± 22.268	1321.978 ± 1.269	11 859.000 ± 46.573
ping float[25600]	112 498.571 ± 648.326	112 744.286 ± 623.054	113 907.143 ± 488.166	15 7030.000 ± 5639.080
ping double[100]	1759.384 ± 23.667	1655.096 ± 1.407	1330.038 ± 19.045	11 875.714 ± 85.667
ping double[12800]	112277.143 ± 373.773	112 500.000 ± 368.743	110 715.714 ± 431.537	18 3528.571 ± 7423.725
ping null	887.370 ± 91.329	980.171 ± 12.019	644.266 ± 0.569	11 648.889 ± 79.505
Non-standard argument types				
ping DMatrix (1024, 1024)	9216571.429 ± 73744.457	not supported	9134000.000 ± 27578.460	10328000.000 ± 175468.353
ping DMatrix (2048, 2048)	36843857.143 ± 85435.738	not supported	36390428.571 ± 109556.062	40476625.000 ± 343571.294
ping Obj4	1148.662 ± 25.494	not supported	1184.615 ± 29.171	13728.571 ± 716.069
ping (Object)IObj4	1115.359 ± 70.439	not supported	1598.169 ± 1.412	16562.857 ± 977.967
ping Obj32	1494.475 ± 63.761	not supported	1479.482 ± 20.564	14801.429 ± 1233.015
ping Tree(1)	1175.644 ± 53.051	not supported	1282.051 ± 0.000	13838.571 ± 746.371
ping Tree(3)	1538.935 ± 81.372	not supported	1559.659 ± 15.109	13682.857 ± 484.995
ping Tree(9)	28505.714 ± 274.427	not supported	27855.714 ± 198.915	177745.714 ± 4935.260

The argument DMatrix has a double[][] array of the given dimensions. Obj32 is an object that has 32 int values, whereas Obj4 has 4 int values. IObj4 extends Obj4 without any additional attributes. Tree is a balanced binary tree of the given number of objects each of which holds four ints. The non-standard types of object invocation parameters were only made available in the calling (importing) node. The called (exporting) node dynamically downloads these types to deserialize the received arguments.

the system will be perceived by the user as sluggish [64]. Furthermore, although we have not tested it in our current experiments, based on the findings of [52], we speculate that use of the tuple space paradigm for internode communication would perform better than XML-based protocols such as UPnP and SOAP over HTTP due to the verbosity and overhead of XML parsing.

6.3. Portability

The portability of our conceptual framework stems largely from the properties of the underlying GVDS concept whose instantiations can differ in any of the following aspects [34]: (1) the chosen data structure; (2) the supported operations on that data structure; (3) the rules for dividing and merging the individual data structures. Together, these characteristics make the GVDS concept applicable to a wide range of distributed environments and application domains including large, wired, wireless and ad-hoc environments [34]. In addition, the proposed conceptual framework provides flexibility in terms of the chosen interaction model and communication protocol. The D-OSGi variations described in Section 5.2 demonstrate the feasibility and portability of the proposed conceptual framework.

6.4. Adaptability

Our conceptual framework and architecture target pervasive environments characterized by device and user mobility. Pervasive applications will need to adapt to changes in resource availability and network topology. Adaptability is needed at two levels: (1) at the individual node level, for example, in response to resource shortage or changed user preferences; (2) at the network level in response to topological changes such as device mobility or a network disconnection. The proposed conceptual framework and its instantiation supports adaptability at these two levels of abstraction. First, at the level of individual devices, OSGi technology provides a *dynamic* application model that allows an application to adapt by changing both its structure and logic. This capability follows from the two service-oriented properties of *dynamicity* (a service provider can offer or withdraw a service at any time and likewise a client can bind or unbind to a service at any time) and *substitutability* (contract-based relations allowing service substitution as long as contract is satisfied) [10]. Second, at the network level, the conceptual framework supports different logical mobility paradigms that allow an application to adapt to changes in network topology.

6.5. OSGi platform as an agent platform

In supporting OSGi service and bundle mobility, we have taken the first but major step towards service-oriented OSGi-based MA platforms. New agent systems can be built and existing ones modified to take advantage of the OSGi platform's benefits which include:

- *Safety and privacy*. An MA platform must ensure that a hostile agent will not compromize the running system. By utilizing OSGi platform's modularity, security, and access control mechanisms, MAs can be safely integrated in the destination host.
- *Interoperability, interaction and communication*. A number of efforts have been undertaken on the interoperability of MA platforms to allow agents developed for a certain platform to be migrated and executed in a different platform (see for example, MASIF [72] and Kalong [46]). We suggest that adopting the OSGi middleware as an agent platform and runtime will help address this interoperability issue. It will also bring many benefits, e.g. modularity, to both the agent platform implementations and the agents themselves. For example, considering a bundle as an agent packaging mechanism will provide a well defined life cycle model for such agents and will also simplify agent integration in the destination nodes. Regarding agent interaction and communication, we argue that the service-oriented properties underpinning the OSGi platform make it ideal choice for agent interaction and resource finding. Thus, the OSGi registry plays the role of the Directory Facilitator as defined by the Foundation for Intelligent Physical agents (FIPA)[12]. FIPA is an IEEE Computer Society standards organization promoting agent-based technology and has defined many standards covering various aspects of MA technology. Service-orientation promotes decoupling and substitutability which are very useful for agent interaction. This does not preclude the use of ACLs on top of service-oriented interaction patterns where it is possible to use service-orientation to locate other agents then use an ACL to communicate with those agents. In particular, we suggest that the service-oriented interaction pattern offers a practical solution to the *open channel* problem [32] which is concerned with the question 'what happens to the open channel between two communicating agents when one of the agents decide to migrate'. Adopting a service-oriented interaction model, each agent is implemented with the assumption that the other agent can be withdrawn/depart at any time. Therefore, the channel can be safely closed and it is then the agents' responsibility to reopen the channel when communication resumes.
- *Agent autonomy*. To realize the autonomy property, many agent system implementations, e.g. JADE[13] [73] run each agent in its own thread of control. Agents in our conceptual framework can be implemented as OSGi bundles which provides them with their own thread of execution that is consistent with other agent systems.
- *Generality*. OSGi technology is being used in a wide range of devices from resource constrained to large enterprise servers. Adopting the OSGi platform as an gent platform brings opportunities for agents that can migrate and communicate across these different and diverse devices.
- *Flexibility*. Silva *et al.* [32] suggest that an MA system must be protocol independent where agent migration can be accomplished using TCP/IP, HTTP or any other transport protocol. Unlike many existing MA systems, our proposed OSGi-based mobility platform together with the flexibility of the conceptual framework provide such protocol independence.

In general, an OSGi-based MA platform will benefit from all the characteristics of the OSGi middleware in terms of, for example, security and access control, modularity, robustness, portability, service-orientation benefits, support for remote agent management and the ability to ensure that all the agent's dependencies including dependencies on the execution platform are satisfied before the agent is allowed to execute.

Recently, FIPA has undertaken a number of activities with the aim of moving standards for agents and agent-based systems into the wider context of software development. FIPA suggests that agent technology needs to work and integrate with non-agent technologies. In particular, FIPA is currently investigating the combination of agent technology with service-oriented architectures and has set up the Agent and Web Services Interoperability (AWSI) Working Group to oversee this effort. FIPA suggests that agents and multi-agent

[12]www.fipa.org/.

[13]http://jade.tilab.com/.

systems can benefit from this combination where agents can be used for web service discovery, consumption, composition and implementation. They also suggest that web services and multiagent systems bear certain similarities, such as a component-like behaviour, that can help to make their integration much easier. We believe that our work in supporting agent concepts on top the service-oriented OSGi platform represents a useful contribution to this research area especially in conjunction with tools like Apache Axis (see Section 7).

7. RELATED WORK

7.1. Distributed OSGi solutions

Wu *et al.* [14] describe a P2P smart home architecture based on OSGi technology and MAs. Their proposed architecture supports the migration of MAs between OSGi-enabled devices to automate and perform tasks. The idea is to encode the agent's plans, behaviour and used resources into a special XML-based scripting documents called MASML that can be pushed and pulled between OSGi devices via Web Services. MASML documents use European Computer Manufacturers Association[14] (ECMA) scripts to encode an agent's logic. A Web Services bundle is pre-installed in each OSGi device to advertise available agents. When an agent document is received by an OSGi device, a special interpreter is used to interpret the encoded tasks and execute them. Execution results are then written back to the same MASML document which can then be returned to the original source device or alternatively it can be migrated to a third device if further tasks need to be performed. This approach does not address service distribution where it is not currently possible to invoke services remotely from connected OSGi devices (although it can be potentially supported via the same web services approach). Our proposed conceptual framework and architecture supports a similar P2P inter-OSGi interaction via MAs that are manifested as OSGi services and bundles. In addition, the conceptual framework supports service distribution and mobility between OSGi nodes both within and across devices. Furthermore, our approach is more consistent with the OSGi application model and does not require special encoding and interpretation steps that complicate application development.

R-OSGi [52, 54, 74] is a research project that aims to provide a lightweight solution to distributed OSGi services. R-OSGi uses the SLP to *discover* and *advertise* remote OSGi services. Using SLP, a device is able to search its environment for advertised services. The motivation behind using the SLP protocol for service advertisement and discovery is described in [54]. R-OSGi supports weak mobility of bundles and services but does not support the software agent abstraction and does not distinguish between migration and remote cloning. In addition, only a pull style of mobility is supported. At the application design level, R-OSGi supports the client–server paradigm. When a service proxy is offered with the destination OSGi device, it forwards all calls to the original service in the source device. For efficiency, these remote calls are modelled using R-OSGi specific messaging protocol although they behave just like any call to a local service. A limited form of REV is supported via the concept of a *smart proxy* which is a normal abstract Java class that is transferred to the destination device as part of the exchanged service properties. All abstract methods of this abstract class are treated as remote method calls and forwarded to the original service in the source device whereas all non-abstract methods are treated as local and invoked on the destination device. R-OSGi supports a resource rebinding strategy wherein a mobile entity can search for and rebind to required resources that are present in the destination device. We argue that different applications will have different requirements and therefore a distribution extension must be flexible to support the different mobility models and resource relocation strategies based on application requirements (customizability). Unlike our proposed solution, R-OSGi lacks such customizability.

As of R-OSGi's latest release 0.6.5, R-OSGi provides *limited* support for method invocation arguments and return values of non-standard types. If a type is reachable from the remote service's interface (i.e. it belongs to the class closure of the service interface), that type will be injected and transferred to the destination node. However, reachable types that are imported from other bundles will *not* be injected. This behaviour is justified because these imported types are considered as part of the required environment and therefore should not be injected for transfer. However, if these types are missing in the destination node, the service can not be offered at destination and hence distribution fails. In addition, R-OSGi does not address the case when the argument type is available in the destination node but not the source node, for example, when the type of invocation argument is a subtype of that of the formal method parameter (R-OSGi type injection is always from source node to destination node). From our experience with R-OSGi and based on a study of its source code, R-OSGi does *not* support remote invocation arguments and return values of *non-standard* types even when such types are available in the target node. In our experiments we noticed that R-OSGi is unable to *deserialize* arguments of non-standard types. R-OSGi gracefully catches the thrown `ClassNotFoundException` to return a `null` object that is then passed as argument to the remote method. Hence, remote methods taking non-standard types as arguments are *always* invoked with `null` as argument, and therefore R-OSGi does not support spontaneous interactions.

The P2Pcomp component model [75] builds on the OSGi platform and the JXTA[15] P2P middleware to support and

[14]www.ecma-international.org.

[15]https://jxta.dev.java.net/.

simplify the development of pervasive computing applications based on spontaneous interaction of mobile peers. Components in P2Pcomp are represented as OSGi bundles offering and using services from each other. P2Pcomp uses the concept of *port*, represented as a Java interface, as an end-point for communication with other components whether they are local or remote. Thus, P2Pcomp provides a unified and transparent approach for component interaction both within and across OSGi frameworks. P2Pcomp distinguishes between three port types: *provides*, *access*, and *uses* ports. Provides ports are used to advertise services that are offered by a P2Pcomponent. Uses ports are used by clients to connect and access the offered services (both local and remote). In other words, both provides and requires ports correspond to the service interface but from the perspectives of the provider and client respectively. Access ports represent the bridges that are used to connect to remote OSGi frameworks. Access ports are protocol independent where any available communication technology can be used. Consequently, both synchronous and asynchronous communication can be supported. P2Pcomp provides two implementations based on JXTA and a proprietary XML-based messaging over UDP/TCP for communication between OSGi framework instances. In the client node, P2Pcomp uses Java's dynamic proxy class to dynamically generate stub objects that implement the required Java interface. In the server node, P2Pcomp interprets the received protocol specific message and uses Java reflection to invoke the appropriate method on the respective service. In comparison with our work, P2Pcomp provides support for OSGi service distribution but not service or bundle mobility. In addition, P2Pcomp does not support spontaneous interactions. We also believe it is possible to use JXTA as an underlying P2P communication infrastructure in a conceptual framework instantiation, a subject for future work.

The service component architecture[16] (SCA) defines a set of specifications for component creation and assembly across programming languages. SCA promotes a service-oriented approach where a component's functionality is offered and consumed through service-oriented interfaces. SCA defines a common XML-based assembly mechanism for combining SCA components independent of their implementation technology. SCA does not address dynamic availability or manage dynamic compositions. Consequently, the open service oriented architecture (OSOA) collaboration, which oversees the SCA and other related specifications, considers SCA, OSGi technology and Spring as complementary technologies [76]. The same white paper discusses ways in which the three technologies can be combined and the resulting benefits of such combination. Of particular relevance to our proposed conceptual framework and architecture is the Newton[17] Project which provides an SCA implementation on the OSGi platform that targets dynamic environments. Newton supports composite SCA components (currently limited to Java-based OSGi components backed by OSGi bundles) that combine components from different JVMs. Newton is able to dynamically manage these composites by dynamically wiring and rewiring their service dependencies as the constituent service offering components come and go. Newton moves code around the network, dynamically installing components on demand and removing them when no longer needed. Newton makes use of OSGi platform's support when wiring SCA composites within a single JVM and Jini technology when wiring SCA composites across different JVMs. Support for dynamic wiring and rewiring of potentially language independent composite components across devices is of particular benefit to ubiquitous environments. Imagine a home environment where various devices offer their functionality as services through SCA compliant components. It is then possible to use SCA technology to create applications that span device boundaries. Infrastructures such as Newton can automatically adapt such applications in response to device mobility, device introduction and removal. However, Newton relies on Jini technology which assumes the existence of a fixed infrastructure i.e. a lookup service, and is tightly coupled to Java RMI. In addition, Newton does not support the mobile software agent concept or distinguish between the different mobility paradigms.

7.2. Other OSGi technology related work

The Spring[18] framework is a Java EE application framework, which integrates with many other frameworks to address the complete Java EE application stack including the Web and back end database layers. At the heart of the Spring framework is an inversion of control (IoC) [77–80] container for developing loosely coupled and easily reconfigurable applications. Spring framework, underlined by the IoC principle, aims to simplify Java EE application development by promoting a Plain Old Java Object (POJO)-based programming model. Recently, the Spring framework has developed a specification for integrating the Spring framework with the OSGi platform. This specification aims to enable the development of Spring-based applications that run on the OSGi platform which supports Spring applications with OSGi modularity so that it is possible to install, uninstall, and update Spring application modules dynamically without the need for restart.

The OSGi Alliance has defined the standard Device Access Specification [12] for interaction between OSGi and non-OSGi devices. This specification adopts an import/export model where devices and services found by native discovery technologies such as UPnP are imported into the OSGi platform as normal OSGi services making them fully accessible to other OSGi entities [6, 12]. Similarly, devices

[16]www.osoa.org/display/Main/Home.
[17]http://newton.codecauldron.org/.
[18]www.springframework.org.

and services that are registered with the OSGi platform can be exported out of the platform so that they become discoverable using native discovery technologies.

Bottaro *et al.* [8] discuss the integration of various distributed technologies over the OSGi platform in the context of home environments. They propose bridges between technologies through *Discovery Base Drivers* and *Refinery Drivers* which are two concepts that have been previously discussed by the standard OSGi Device Access Specification. According to [8], a discovery base driver manifested as an OSGi bundle runs in the device and leverages its specific protocol stack to detect and react to network events such as device arrivals and departures by dynamically populating the local OSGi registry with proxy services that represent the remote devices. Service proxies registered by discovery base drivers provide *generic APIs* that allow a developer to interact with the remote device. However, developers still have to map the device specific protocols to this API which means that such devices can only be accessed at a very low level of transparency. Refining Drivers alleviate this problem by detecting the registration of the generic proxy services to automatically register new proxy services that provide higher levels of transparency for accessing those remote devices. Our proposed approach is similar in so far as it automatically generates proxy services for interaction with remote devices but at the same time it differs in many respects. First, the authors' approach based on discovery and refinery drivers describes a general solution to interaction between OSGi devices and other non-OSGi devices but fails to mention how the same approach can be applied for OSGi–OSGi interaction which is the focus of our work. The two works therefore complement each other. Second, their approach is limited to client–server interactions whereas ours supports all the mobility paradigms. On the other hand, it is possible to use the base discovery and refinery drivers in implementations of our conceptual framework. For example, it is possible to create drivers that announce/detect the connectivity of OSGi devices using specific protocols. When sharing the OSGi services, refinery drivers can be used to transparently map the shared services into higher level services that provide simpler and more developer friendly APIs that can also overcome the interface fragmentation problem [8].

Apache Axis[19] is a project that aims to simplify Web Service development. Axis implements the SOAP protocol and shields the developer from directly dealing with SOAP and the Web Service Description Language. For example, Axis can automatically encode/decode web service requests and responses to/from SOAP XML messages. Axis greatly simplifies Web Service development where it can automatically expose POJOs as Web Services. The Knopflerfish project provides an OSGi-based Axis implementation that can be used to expose OSGi services as Web Services. Axis can be used as an underlying technology when realizing the proposed conceptual framework.

The Bnd[20] tool is a tool that helps create and diagnose OSGi R4 bundles. It can wrap JAR files into OSGi bundles, create OSGi bundles from a given Bnd specification, and verify the validity of bundle manifest entries. Bnd recognizes Java classes and packages using the classpath to locate, analyse, construct and verify the generated bundle. A Bnd specification uses attributes and directives very similar to those found in an OSGi bundle manifest header. The Bnd specification format is simple but powerful enough to allow fine control of bundle generation. Thus, the Bnd tool can be useful for generating the proxy bundles used to integrate the mobile entity in the destination node. Although it provides a command line interface for invoking the tool, Bnd currently lacks an API that can be used for bundle generation. As future work, we plan to investigate the use of Bnd for proxy bundle generation.

7.3. GVDS and tuple space implementations

Lime, XMIDDLE [81] and PeerWare [82] are all middleware systems that implement the GVDS concept. These systems differ in terms of their chosen data structure, and rules for sharing. Lime uses a *tuple space* data structure that is partitioned among the different nodes. The content of these individual tuple spaces are then transiently shared when their owning nodes become connected. PeerWare uses *tree* data structures whose content are transiently shared by merging them with trees from other nodes when these nodes are connected. XMIDDLE uses *XML documents* as a data structure that is transiently shared by combining them with other XML documents from connected nodes.

Various adaptations of the original Lime middleware have been developed that target different application domains. For example, *TinyLime* [83, 84] is an extension of Lime for sensor network environments where the basic Lime model is adapted to allow access to sensor data. *TeenyLime* [85] is another adaptation of Lime to sensor and actuator networks. The target application area of this adaptation is wireless sensor networks (WSN) containing both actuators and triggering sensors. TeenyLime aims to simplify the development of WSN applications in such scenarios. Carbunar *et al.* [86] revisit the Lime model with the aim of simplifying the model. They identify the core components of the Lime model and factor it out into a new model which they call *CoreLime*. The same authors suggest that the proposed CoreLime model offers fine-grained access control and can better scale to large configurations.

Limone [87] and MESHMdl [88] are other lightweight implementations of tuple space-based coordination middleware targeting mobile devices. MESHMdl supports the GVDS concept via what it calls a *Xector* model, which is

[19]http://ws.apache.org/axis/.

[20]www.aqute.biz/Code/Bnd.

essentially a mechanism for selectively pushing and pulling tuples (modelled as Entries in MESHMdl) between the different nodes. Interestingly, MESHMdl claims superior tuple space performance (even on mobile devices) than other tuple space implementations including Lime. As future work we plan to investigate these alternative GVDS implementations as basis for a D-OSGi implementation which may also improve the performance of the current prototypes. Tuple space systems as realized by the Lime middleware are shown to provide useful features for communication in mobile environments. Other tuple space systems include TSpaces [89] and JavaSpaces [90] which are more suited to resource rich devices in nomadic and fixed network settings as opposed to mobile ad-hoc environments. Each of these systems can be utilized to realize instances of the proposed conceptual framework targeting specific domains.

7.4. Mobile agent platforms for resource constrained devices

Of the many available agent platforms, very few are capable of running on mobile and embedded devices (see for example, [91]). The Lightweight and Extensible Agent Platform (LEAP) [92] is an adaptation of the JADE agent platform for mobile and resource constrained devices. LEAP is deployed on the basis of *profiles* that match specific device capabilities. LEAP is fully FIPA compliant and supports different protocols for communication between agents residing in different nodes, namely, Internet Inter-ORB Protocol, Java RMI and a proprietary TCP/IP protocol for wired and wireless networks. Our OSGi-based agent platform also targets resource constrained devices but it is not currently FIPA compliant.

8. CONCLUSION AND FUTURE WORK

This paper described two extensions to the OSGi platform: (1) support for spontaneous interoperability between OSGi nodes in pervasive environments; (2) support for logical mobility. These two extensions aim to address the pervasive requirements of spontaneous interoperability, mobility and software adaptability. Supporting logical mobility also facilitates the adoption of the mobile software agent concept for home automation, software adaptation, device interaction and collaboration. The two extensions are supported through a common framework which, similar to other OSGi distribution solutions, blurs the distinction between local and remote services. Remote services can be accessed as if they were local which greatly simplifies application development. A novel aspect of our approach lies in the application of the GVDS concept to provide transparent context maintenance for OSGi devices. The conceptual framework introduces a virtual shared OSGi registry whose content is automatically and dynamically adjusted to reflect changes in the system and the mobile environment.

Instantiations of our conceptual framework facilitate inter-device service composition in mobile ad-hoc environments. We also believe that instantiations of our conceptual framework will facilitate and simplify development of Newton like infrastructures. A unique potential of our conceptual framework is the leveraging of agent technology and mobility support for *proactive* service composition. By proactive service composition we mean the ability to exploit agent knowledge and beliefs to deduce and reason about service requirements which can then be obtained on demand as opposed to waiting for those services to become available. In future work, we plan to investigate and adapt the ideas put forward by [93] to OSGi technology.

An interesting future work is better integration of the D-OSGi API into the OSGi specification by having D-OSGi wrap the `BundleContext` according to the decorator design pattern [42] to provide the bundle with a special `DOSGiBundleContext` that adds an API for mobility and distribution. For example, the decorator could define a method **public** `ServiceRegistration registerRemoteService(String clazz, Object service, Dictionary properties, String remoteLocation IPAddress)` that registers the specified service with the OSGi registry of the specified remote location. Another interesting future work is to discuss and compare our OSGi-based agent platform in relation to other existing MA systems using a reference model such as that described in [32]. A related task is supporting and complying with FIPA specifications.

ACKNOWLEDGEMENTS

We would like to thank the four reviewers for their valuable feedback and detailed suggestions, which have helped to improve this paper. Our gratitude is also due to the editor for giving us an opportunity to revise our paper. We would also like to thank the developers in the osgi-dev@www2.osgi.org mailing list for their help and valuable feedback. This work is supported by EPSRC and NXP Semiconductors.

FUNDING

This work is supported by the Engineering and Physical Sciences Research Council (EPSRC) and NXP Semiconductors.

REFERENCES

[1] Weiser, M. (1993) Some computer science issues in ubiquitous computing. *Commun. ACM*, **36**, 74–84.

[2] Niemelä, E. and Latvakoski, J. (2004) Survey of Requirements and Solutions for Ubiquitous Software. *MUM '04: Proc. 3rd Int Conf. Mobile and Ubiquitous Multimedia*, College Park, Maryland, October 27–29, pp. 71–78. ACM Press, NY, USA.

[3] Satyanarayanan, M. (2001) Pervasive computing: vision and challenges. *IEEE Pers. Commun.*, **8**, 10–17.

[4] Da Costa, C.A., Yamin, A.C. and Geyer, C.F.R. (2008) Toward a general software infrastructure for ubiquitous computing. *IEEE Pervasive Comput.*, **7**, 64–73.

[5] Kindberg, T. and Fox, A. (2002) System software for ubiquitous computing. *IEEE Pervasive Comput.*, **1**, 70–81.

[6] Dobrev, P., Famolari, D., Kurzke, C. and Miller, B. (2002) Device and service discovery in home networks with OSGi. *IEEE Commun. Mag.*, **40**, 86–92.

[7] Valtchev, D. and Frankov, I. (2002) Service gateway architecture for a smart home. *IEEE Commun. Mag.*, **40**, 126–32.

[8] Bottaro, A., Gerodolle, A. and Lalanda, P. (2007) Pervasive Service Composition in the Home Network. *AINA '07: Proc. 21st Int. Conf. Advanced Networking and Applications*, Washington, DC, USA, May 21–23, pp. 596–603. IEEE Computer Society.

[9] Satyanarayanan, M. (1996) Accessing information on demand at any location: mobile information access. *IEEE Pers. Commun.*, **3**, 26–33.

[10] Papazoglou, M. (2003) Service-oriented Computing: Concepts, Characteristics and Directions. *Proc. Fourth Int. Conf. Web Information Systems Engineering*, Rome, Italy, December 10–12 pp. 3–12. IEEE Computer Society.

[11] Hall, R. and Cervantes, H. (2004) Challenges in building service-oriented applications for OSGi. *IEEE Commun. Mag.*, **42**, 144–9.

[12] The OSGi Alliance, (2005) *OSGi Service Platform Core Specification and Service Compendium, Release 4.1*. OSGi Alliance. CA, USA. http://www.osgi.org/.

[13] Li, X. and Zhang, W. (2004) The design and implementation of home network system using OSGi compliant middleware. *IEEE Trans. Consum. Electron.*, **50**, 528–534.

[14] Wu, C.-L., Liao, C.-F. and Fu, L.-C. (2007) Service-oriented smart-home architecture based on OSGi and mobile-agent technology. *IEEE Trans. Syst., Man Cybern. C Appl Rev.*, **37**, 193–205.

[15] Lee, C., Nordstedt, D. and Helal, S. (2003) Enabling smart spaces with OSGi. *IEEE Pervasive Comput.*, **2**, 89–94.

[16] Zahariadis, T. and Pramataris, K. (2002) Multimedia home networks: standards and interfaces. *Comput. Stand. Interfaces*, **24**, 425–35.

[17] Tanenbaum, A. and van Steen, M. (2002) *Distributed Systems: Principles and Paradigms*. Prentice Hall, Upper Saddle River, NJ.

[18] Milojicic, D.S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S. and Xu, Z. (2002) Peer-to-peer Computing. Technical Report HPL- 2002-57. HP Laboratories, Palo Alto.

[19] Birrell, A.D. and Nelson, B.J. (1984) Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, **2**, 39–59.

[20] Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec, A.-M. (2003) The many faces of publish/subscribe. *ACM Comput. Surv.*, **35**, 114–131.

[21] Hayes, C.C. (1999) Agents in a nutshell—a very brief introduction. *IEEE Trans. Knowl. Data Eng.*, **11**, 127–132.

[22] Fuggetta, A., Picco, G. and Vigna, G. (1998) Understanding code mobility. *IEEE Trans. Softw. Eng.*, **24**, 342–61.

[23] Cervantes, H. and Hall, R. (2003) Automating Service Dependency Management in a Service-oriented Component Model. *Proc. Sixth Component-based Software Engineering Workshop, May*, pp. 91–96. Portland, United States.

[24] Cervantes, H. and Hall, R.S. (2004) Autonomous Adaptation to Dynamic Availability Using a Service-oriented Component Model. *ICSE '04: Proc. 26th Int. Conf. Software Engineering*, Washington, DC, USA, May 23–28, pp. 614–623. IEEE Computer Society.

[25] Baumann, J., Hohl, F., Rothermel, K. and Straer, M. (1998) Mole—concepts of a mobile agent system. *World Wide Web*, **1**, 123–137.

[26] Pham, V.A. and Karmouch, A. (1998) Mobile software agents: an overview. *IEEE Commun. Mag.*, **36**, 26–37.

[27] Ghezzi, C. and Vigna, G. (1997) Mobile Code Paradigms and Technologies: a Case Study. In Rothermel, K. and Popescu-Zeletin, R.(eds) *Proc. First Int. Workshop on Mobile Agents*, Berlin, Germany, April 07–08, *Lecture Notes in Computer Science*, Vol. 1219, Springer, London, pp. 39–49.

[28] Carzaniga, A., Picco, G. and Vigna, G. (1997) Designing Distributed Applications with Mobile Code Paradigms. *Proc. 19th Int. Conf. Software Engineering*, Boston, Massachusetts, United States, May 17–23, pp. 22–32. ACM, New York.

[29] Griss, M.L. and Pour, G. (2001) Accelerating development with agent components. *Computer*, **34**, 37–43.

[30] Horvat, D., Cvetkovic, D., Milutinovic, V., Kocovic, P. and Kovacevic, V. (2001) Mobile agents and java mobile agents toolkits. *Telecommun. Syst. Model. Anal., Des. Manag.*, **18**, 271–287.

[31] Labrou, Y., Finin, T. and Peng, Y. (1999) Agent communication languages: the current landscape. *IEEE Intellig. Syst.*, **14**, 45–52.

[32] Silva, A., Romao, A., Deugo, D. and Mira da Silva, M. (2001) Towards a reference model for surveying mobile agent systems. *Auton. Agents Multi-Agent Syst.*, **4**, 187–231.

[33] Baldi, M. and Picco, G.P. (1998) Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. *Proc 20th Int. Conf. Software Engineering*, Kyoto, Japan, April 19–25, pp. 146–155. IEEE Computer Society, Washington, DC.

[34] Murphy, A., Picco, G. and Roman, G. (2002) On global virtual data structures. In Marinescu, D. and Lee, C. (eds) *Process Coordination and Ubiquitous Computing*, pp. 11–29. CRC Press.

[35] Baude, F., Caromel, D., Huet, F. and Vayssiere, J. (2000) Communicating Mobile Active Objects in Java. In Bubak, M.,

Williams, R., Afsarmanesh, H. and Hertzberger, L.O. (eds) *Proc. 8th Int. Conf. High-performance Computing and Networking*, May 08–10, *Lecture Notes in Computer Science*, Vol. 1823. Springer, London, pp. 633–643.

[36] Schmidt, D.C. and Cranor, C.D. (1996) Halfsync/ half-async: An Architectural Pattern for Efficient and Well-structured Concurrent i/o. *Pattern Languages of Program Design 2*, pp. 437–459. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.

[37] Scott, M. (1999) *Programming Language Pragmatics* pp. 20–21. Morgan Kaufmann.

[38] Walker, E., Floyd, R. and Neves, P. (1990) Asynchronous Remote Operation Execution in Distributed Systems. *Proc. 10th Int. Conf. Distributed Computing Systems*, Paris, France, May 28–June 1, pp. 253–259.

[39] Pitt, E. and McNiff, K. (2001) *Java.rmi: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.

[40] Waldo, J. (1998) Remote procedure calls and java remote method invocation. *IEEE Concurrency (see also IEEE Parallel & Distributed Technology)*, **6**, 5–7.

[41] The OSGi Alliance (2004) Listeners Considered Harmful: The Whiteboard Pattern. Technical Whitepaper, Revision 2.0. [Online]. http://www.osgi.org/documents/osgi_technology/whiteboard.pdf.

[42] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.

[43] Allard, J., Chinta, V., Gundala, S. and Richard, I.G.G. (2003) Jini Meets UPnP: An Architecture for Jini/UPnP Interoperability. *Proc. 2003 Symp. Applications and the Internet*, Orlando, FL, USA, January 27–31, pp. 268–275. IEEE Computer Society, Washington, DC.

[44] Guttman, E. and Kempf, J. (1999) Automatic Discovery of Thin Servers: SLP, Jini and the SLP–Jini Bridge. *Proc. 25th Annual Conf. IEEE Industrial Electronics Society (IECON'99)*, San Jose, CA, USA, November 29–December 03, pp. 722–727.

[45] Picco, G.P. (1998) μcode: A Lightweight and Flexible Mobile Code Toolkit. In Rothermel, K. and Hohl, F. (eds) *Proc. Second Int. Workshop on Mobile Agents*, Berlin, Germany, *Lecture Notes in Computer Science*, Vol. 1477. Springer, London, UK, pp. 160–171.

[46] Braun, P., Trinh, D. and Kowalczyk, R. (2005) Integrating a New Mobility Service into the Jade Agent Toolkit. In Karmouch, A. and Pierre, S. (eds) *Proc. Second Int. Workshop Mobility Aware Technologies and Applications (MATA 2005)*, Montreal, Que, Canada., October 17–19, *Lecture Notes in Computer Science*, Vol. 3744. Springer, Berlin, pp. 354–63.

[47] Cugola, G., Nitto, E.D. and Fuggetta, A. (2001) The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, **27**, 827–850.

[48] Huang, Y. and Garcia-Molina, H. (2004) Publish/subscribe in a mobile environment. *Wirel. Netw.*, **10**, 643–652.

[49] Cugola, G. and Jacobsen, H.-A. (2002) Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, **6**, 25–33.

[50] Mascolo, C., Capra, L. and Emmerich, W. (2002) Mobile computing middleware. *Advanced Lectures on Networking: Networking 2002 Tutorials*, Vol. 2497, pp. 20–58.

[51] Kendall, S.C., Waldo, J., Wollrath, A. and Wyant, G. (1994) A Note on Distributed Computing. Technical Report. Sun Microsystems Inc., Mountain View, CA, USA.

[52] Rellermeyer, J.S., Alonso, G. and Roscoe, T. (2007) ROSGi: Distributed Applications Through Software Modularization. *Proc. ACM/IFIP/USENIX 8th Int. Middleware Conf.*, Newport Beach, CA, USA, November 26–30, *Lecture Notes in Computer Science*, Vol. 4834. Springer, Berlin.

[53] Johansen, D., van Renesse, R. and Schneider, F.B. (1999) Operating system support for mobile agents. *Mobility: Processes, Computers, and Agents*, pp. 557–563. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

[54] Rellermeyer, J.S. and Alonso, G. (2007) Services Everywhere: OSGi in Distributed Environments. *Presented in EclipseCon 2007*, March 5–8, Santa Clara, C .

[55] Offermans, M. (2005) Automatically managing service dependencies in OSGi (online). http://www.osgi.org/news_events/documents/.

[56] Murphy, A.L., Picco, G.P. and Roman, G.-C. (2006) LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, **15**, 279–328.

[57] di Laurea di, , T. (2004) LIME II. Phd Thesis. Politecnico Di milano.

[58] Gelernter, D. (1985) Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, **7**, 80–112.

[59] Lavender, R.G. and Schmidt, D.C. (1996) Active Object: An Object Behavioral Pattern for Concurrent Programming. *Pattern Languages of Program Design 2*, pp. 483–499. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA.

[60] Carzaniga, A., Rosenblum, D. and Wolf, A. (2001) Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, **19**, 332–383.

[61] Cilia, M., Fiege, L., Haul, C., Zeidler, A. and Buchmann, A.P. (2003) Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware. *Proc. 2nd Int. Workshop on Distributed Event-based Systems (DEBS '03)*, San Diego, CA, June 08–08, pp. 1–8. ACM, New York, NY, USA.

[62] Caporuscio, M., Carzaniga, A. and Wolf, A.L. (2003) Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. Softw. Eng.*, **29**, 1059–1071.

[63] Muhl, G., Ulbrich, A. and Herrman, K. (2004) Disseminating information to mobile clients using publish-subscribe. *IEEE Internet Comput.*, **8**, 46–53.

[64] Johanson, B. and Fox, A. (2004) Extending tuplespaces for coordination in interactive workspaces. *J. Syst. Softw.*, **69**, 243–266.

[65] Barrett, D.J., Clarke, L.A., Tarr, P.L. and Wise, A.E. (1996) A framework for event-based software integration. *ACM Trans. Softw. Eng. Methodol.*, **5**, 378–421.

[66] Rosenblum, D.S. and Wolf, A.L. (1997) A Design Framework for Internet-scale Event Observation and Notification. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, **22**, 344–360.

[67] Hagimont, D. and Ismail, L. (1999) A Performance Evaluation of the Mobile Agent Paradigm. In Berman, A.M. (ed) *Proc. 14th ACM SIGPLAN Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, Denver, Colorado, United States, November 01–05, pp. 306–313. ACM, New York, NY, USA.

[68] Straber, M. and Schwehm, M. (1997) A Performance Model for Mobile Agent Systems. In Arabnia, H. (ed) *Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA97)*, Las Vegas, July, pp. 1132–1140. Computer Science Research, Education, and Application Technology (CSREA), New York, USA.

[69] Chia, T.-H. and Kannapan, S. (1997) Strategically mobile agents. In Rothermel, K. and Popescu-Zeletin, R. (eds) *Proc First Int. Workshop on Mobile Agents (MA '97)*, London, UK, April 07–08, *Lecture Notes in Computer Science*, Vol. 1219, Springer, London, pp. 149–161.

[70] Philippsen, M., Haumacher, B. and Nester, C. (2000) More efficient serialization and rmi for java. *Concurrency Pract. Exp.*, **12**, 495–518.

[71] Padovitz, A., Zaslavsky, A. and Loke, S.W. (2003) Awareness and agility for autonomic distributed systems: platform- independent and publish-subscribe event-based communication for mobile agents. *Proc. 14th Int. Workshop on Database and Expert Systems Applications (DEXA'03)*, Prague, Czech Republic, September 1–5, pp. 669–73. IEEE Computer Society.

[72] Milojicic, D.S. *et al*. (1999) MASIF: The OMG Mobile Agent System Interoperability Facility. In Rothermel, K. and Hohl, F. (eds) *Proc. Second Int. Workshop on Mobile Agents (MA '98)*, *Lecture Notes in Computer Science*, Vol. 1477. Springer, London, UK, pp. 50–67.

[73] Bellifemine, F., Poggi, A. and Rimassa, G. (2001) JADE: A FIPA2000 Compliant Agent Development Environment. *Proc. fifth Int. Conf. Autonomous Agents (AGENTS '01)*, Montreal, Quebec, Canada, May 28–June 1, pp. 216–217. ACM, New York, NY, USA.

[74] Rellermeyer, J.S. (2006) FlowSGi: A framework for dynamic fluid applications. Master's Thesis. ETH Zurich.

[75] Ferscha, A., Hechinger, M., Mayrhofer, R. and Oberhauser, R. (2004) A light-weight component model for peer-to-peer applications. *Proc. 24th Int. Conf. Distributed Computing Systems Workshops—W7: EC (ICDCSW'04)*, Hachioji, Tokyo, Japan, March 23–24, pp. 520–527. IEEE Computer Society, Washington, DC, USA.

[76] The Open Service Oriented Architecture (OSOA) Collaboration (2007). Power combination: SCA, OSGi and Spring. white paper (online). http://www.osoa.org/display/Main/SCA+Resources.

[77] Fowler, M. (2004) Inversion of control containers and the dependency injection pattern (online). http://www.martinfowler.com/articles/injection.html.

[78] Mathew, S. (2005). Examining the validity of inversion of control (online). http://www.theserverside.com/tt/articles/article.tss?l=IOCandEJB.

[79] Johnson, R. (2005). Introduction to the spring framework (online). http://www.theserverside.com/tt/articles/article.tss?l=SpringFramework.

[80] Weiskotten, J. (2006) Dependency injection: designing loosely coupled and testable objects. *Dr. Dobb's J.*, **31**, 10–15.

[81] Mascolo, C., Capra, L., Zachariadis, S. and Emmerich, W. (2002) XMIDDLE: a data-sharing middleware for mobile computing. *Wirel. Pers. Commun.*, **21**, 77–103.

[82] Cugola, G. and Picco, G.P. (2002) Peer-to-peer for Collaborative Applications. *Proc. 22nd Int. Conf. Distributed Computing Systems (ICDCSW '02)*, Vienna, Austria, July 02–05, pp. 359–364. IEEE Computer Society, Washington, DC, USA.

[83] Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L. and Picco, G.P. (2005) Mobile data collection in sensor networks: the tinylime middleware. *Pervasive Mob. Comput.*, **1**, 446–469.

[84] Curino, C., Giani, M., Giorgetta, M., Giusti, A., Murphy, A.L. and Picco, G.P. (2005) Tinylime: Bridging Mobile and Sensor Networks Through Middleware. *Proc. Third IEEE Int. Conf. Pervasive Computing and Communications (PERCOM '05)*, Kauai Island, HI, USA, March 08–12, pp. 61–72. IEEE Computer Society, Washington, DC, USA.

[85] Costa, P., Mottola, L., Murphy, A.L. and Picco, G.P. (2006) Teenylime: Transiently Shared Tuple Space Middleware for Wireless Sensor Networks. *Proc. Int. Workshop Middleware Sensor Networks (MidSens '06)*, Melbourne, Australia, November 28–28, pp. 43–48. ACM Press, New York, NY, USA.

[86] Carbunar, B., Valente, M. and Vitek, J. (2001) Lime Revisited (Reverse Engineering an Agent Communication Model). *Proc. 5th Int. Conf. Mobile Agents (MA 2001)*, December 2–4, *Lecture Notes in Computer Science*, Vol. 2240, pp. 54–69. Springer, Berlin/Heidelberg, Atlanta, GA, USA.

[87] Fok, C.-L., Roman, G.-C. and Hackmann, G. (2004) A Lightweight Coordination Middleware for Mobile Computing. In De Nicola, F.G.R. and Meredith, G. (eds) *Proc. 6th Int. Conf. Coordination Models and Languages (Coordination 2004)*, Pisa, Italy, February 24–27, *Lecture Notes in Computer Science*, Vol. 2949, Springer, Berlin/Heidelberg, pp. 135–151.

[88] Herrmann, K., Mühl, G. and Jaeger, M.A. (2007) Meshmdl event spaces—a coordination middleware for self-organizing applications in ad hoc networks. *Pervasive Mob. Comput.*, **3**, 467–487.

[89] Lehman, T., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B. and Bowman, P. (2001) Hitting the distributed computing sweet spot with TSpaces. *Comput. Netw.*, **35**, 457–472.

[90] Freeman, E., Hupfer, S. and Arnold, K. (1999) *JavaSpaces^{TM} Principles, Patterns, and Practice*. Addison-Wesley, Reading, MA.

[91] Tarkoma, S. and Laukkanen, M. (2002) Supporting Software Agents on Small Devices. *Proc. First Int. Joint Conf. Autonomous Agents and Multiagent Systems: Part 2 (AAMAS*

'02), Bologna, Italy, July 15–19, pp. 565–566. ACM, New York, NY, USA.

[92] Bergenti, F. and Poggi, A. (2002) LEAP: A FIPA platform for handheld and mobile devices. In Meyer, J.C. and Tambe, M.(eds) *Intelligent Agents VIII: Revised Papers from the 8th Int. Workshop on Agent Theories, Architectures, and Languages (ATAL 2001)*, Seattle, WA, USA, August 01–03, 2001, *Lecture Notes in Computer Science*, Vol. 2333, Springer, London, UK, pp. 436–446.

[93] Kalasapur, S., Kumar, M. and Shirazi, B.A. (2007) Dynamic service composition in pervasive computing. *IEEE Trans. Parallel Distrib. Syst.*, **18**, 907–918.