The Journal of Systems and Software 86 (2013) 501–519

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Building ubiquitous computing applications using the VERSAG adaptive agent framework

Kutila Gunasekera [a,*], Arkady Zaslavsky [b], Shonali Krishnaswamy [a], Seng Wai Loke [c]

[a] Faculty of Information Technology, Monash University, Caulfield East, VIC, Australia
[b] ICT Centre, CSIRO, Canberra, ACT, Australia
[c] Department of Computer Science & Computer Engineering, La Trobe University, Bundoora, VIC, Australia

ARTICLE INFO

Article history:
Received 23 March 2012
Received in revised form 10 August 2012
Accepted 17 September 2012
Available online 25 September 2012

Keywords:
Agent architecture
Ubiquitous computing
Mobile agents

ABSTRACT

In this article, we describe a novel approach to build ubiquitous computing applications using adaptive software agents. Towards this, we propose VERSAG, a novel agent framework and architecture which combines agent mobility with the ability to dynamically change the internal structure and capabilities of agents and leads to highly versatile and lightweight software agents. We describe the framework in depth and provide design and implementation details of our prototype implementation. A case study scenario is used to illustrate the functional benefits achievable through the use of this framework in ubiquitous computing environments. Further experimental evaluation confirms the efficiency and feasibility of the VERSAG framework which outperforms traditional mobile agents, and also demonstrates applicability of the proposed framework to agent based systems where varying capabilities are required by agents over their lifecycle.

1. Introduction

Ubiquitous computing applications need to be part of their users' physical environment and seamlessly integrate with their daily activities. In order to achieve these goals, such applications need to possess special characteristics. For example, they have to support heterogeneous devices which are purpose built, battery powered, mobile, have poor processing capacity in comparison to desktop computers, have limited input/output mechanisms, and contain diverse hardware and software (Barton et al., 2004; Saha and Mukherjee, 2003; Satyanarayanan, 2001). In addition, they have to be able to adapt in response to changes in user activity, goals and environment. Banavar and Bernstein (2002, 2004) identify *semantic modelling*, *building software infrastructure*, *developing/configuring applications* and *validating the user experience* to be key challenge areas that have to be addressed in ubiquitous computing. The challenge of building necessary software infrastructure is the focus of our current research. Banavar and Bernstein (2002, 2004) further state that a suitable software infrastructure for ubiquitous computing should be able to unobtrusively determine most relevant tasks to be addressed, find and synthesize applications to fulfil these tasks, be able to migrate if required, robust enough to operate in resource constrained environments, and scalable to support large numbers of devices, users and applications.

While different approaches to the challenge of software infrastructure have been investigated (e.g. autonomic computing and adaptive software (Kephart and Chess, 2003; Salehie and Tahvildari, 2009)), mobile agent technology too is seen as an attractive option (Braun and Rossak, 2004; Cardoso and Kon, 2002; Lange and Oshima, 1999; Satyanarayanan, 2001). Recent works such as the ODDUGI mobile agent platform (Choi et al., 2009), the (Irish) Agent Factory (Muldoon et al., 2007) and MobiSoft (Erfurth et al., 2008) are representative of efforts to harness this suitability. We previously proposed VERSAG (Gunasekera et al., 2009a, 2010) as an approach to harness the potential of mobile agents for building ubiquitous computing applications. VERSAG combines agent mobility with the ability to dynamically change the internal structure and capabilities of agents to generate highly versatile and lightweight software agents. A further contribution of VERSAG is the concept of sharing functional components amongst peer agents as a mechanism to increase efficiency and utility of agents in ubiquitous computing environments.

In this study, we propose, implement and evaluate the VERSAG framework and agent architecture. An in depth description of the framework and details of our prototype design and implementation are presented. As a further contribution, we use a case study scenario of a personal assistant agent to demonstrate the advantages that VERSAG brings to ubiquitous computing applications. We also present experimental results which illustrate the increase in efficiency that can be achieved through VERSAG.

The rest of this article is structured as follows. In Section 2 we describe the VERSAG framework in detail followed by prototype

* Corresponding author. Tel.: +61 423 588770.
E-mail address: kutila@gmail.com (K. Gunasekera).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.09.026

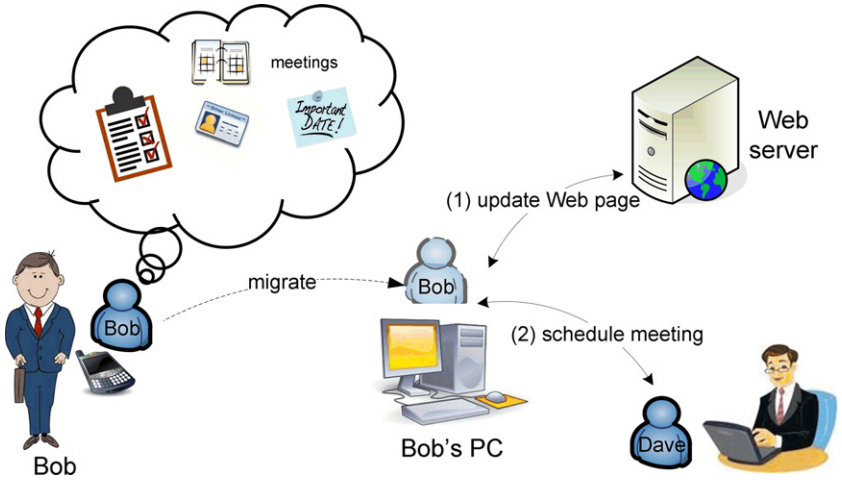

Fig. 1. Scenario of a ubiquitous personal assistant agent.

design and implementation descriptions in Section 3. We present our empirical evaluations in Section 4. Section 5 briefly covers some related work and highlights how VERSAG improves on the state-of-the-art. Section 6 discusses the significance of VERSAG in domains other than ubiquitous computing. We conclude the article in Section 7.

2. The VERSAG agent framework

We start this section with a scenario to motivate the need for VERSAG and then proceed to describe the agent framework.

2.1. Motivating scenario

Let us consider the use of mobile agents as ubiquitous and virtual personal assistants of human users as seen in Erfurth et al. (2008). A personal assistant agent encapsulates a profile of the user including details such as identity, preferences, personal calendar and reminders, and is the user's representative in the virtual pervasive world. As shown in Fig. 1, the agent can "live" on computing devices that its owner uses (e.g. smart phone, tablet computer, desktop PC) and also migrate to other devices when required. It is desirable that this agent is long-lived and capable of carrying out different tasks on behalf of its owner. This in turn implies that the agent should contain a large skill set, which would make it bulky and unsuitable for execution on resource constrained devices and migration over wireless links. It is advantageous in such situations if the agent can adapt itself by dynamically loading and shedding functionality based on its needs. Such an agent could be lightweight when it resides on resource constrained devices (e.g. a smart phone), and when assigned a task, migrate to a more powerful computer, acquire necessary skills and carry out the requested task. Also, with the ability to take up new functionality as needed, the agent could extend itself to take advantage of new applications/resources found on its host devices.

When realising such a solution, several issues related to agent adaptation need to be addressed. First, agent functionality needs to be represented as software components that can be attached to and detached from agents at runtime. Secondly, an agent faced with a task for which it does not have the necessary functionality should be able to search for and obtain components implementing this functionality from nearby agents which possess the same. Thirdly, while working towards its goal, the agent should be able to adapt its activities based on the outcomes of previous activities as well as environmental states. Adaptation here includes agent migration as well as acquiring and discarding components. The adaptation process should take into account multiple criteria including contextual conditions, user preferences, application requirements and available alternatives. Therefore, agents need to possess sophisticated adaptation decision making techniques in order to perform their tasks in a cost-efficient manner. We next describe the VERSAG agent framework which is our proposal to address these challenges.

2.2. Versatile self-adaptive agents

The VERsatile Self-adaptive AGents (VERSAG) framework is centred on a component-based agent architecture that allows agents to dynamically share components with peer agents and adapt based on contextual needs. Application-specific functionality of a VERSAG agent is provided in the form of reusable software components termed *capabilities*. This facilitates an agent to gain diverse behaviours by using appropriate capabilities. The two salient features of the proposed solution are as follows.

- *Simple primitive operations*. An agent in VERSAG is modelled as an active mobile entity with a simple set of primitive operations which allow it to migrate (*move*), search for and acquire capabilities from external sources (*get*), execute capabilities it possesses (*start/stop*), discard unwanted capabilities (*discard*) and terminate itself (*terminate*). An itinerary (Price et al., 2007) in VERSAG defines an agent's migration path and activities in terms of these primitive operations. Agent adaptation too is achieved through these primitive operations and can be either driven by environmental context or triggered by functional requirements.
- *Peer capability sharing*. A VERSAG agent does not depend on a single capability source. Instead, when it does not possess required capabilities, an agent is able to search for, select and acquire appropriate capabilities from peer agents who are willing to share their capabilities (as well as from a centralized capability repository).

The six primitive operations and the peer capability sharing feature form the nucleus of a VERSAG agent as shown in Fig. 2. Around this nucleus is a flexible outer layer where new functionality can be dynamically attached to or detached from the agent in the form of capabilities. This approach allows building complex and sophisticated agents which are also highly versatile.

A VERSAG agent, in addition to dynamically gaining new application-specific functionality, is capable of increasing its autonomy through acquisition of new capabilities which improve its core

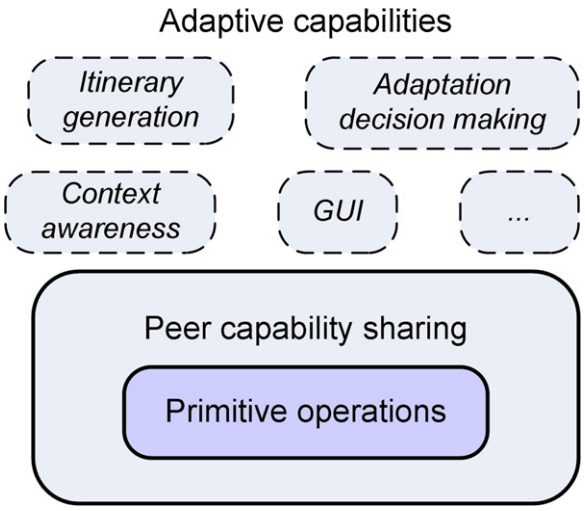

Fig. 2. Overview of VERSAG agent features.

functions such as reasoning ability, context-awareness and capability sharing protocols. For example, an agent can acquire a capability which implements a reinforcement learning strategy (Kaelbling et al., 1996) in order to handle new situations that it encounters. Alternative implementations of basic agent services such as communication and migration too could be implemented as capabilities. This enables an agent to dynamically adapt its behaviours such as communication, migration, capability sharing protocols, sensing and context-awareness based on application and environmental needs.

VERSAG agents are therefore lightweight by default and can dynamically increase/decrease their sophistication based on needs by acquiring and discarding capabilities accordingly. We argue that these features make VERSAG agents especially suited for ubiquitous computing scenarios where heterogeneous environments and rapidly changing requirements are commonplace. We describe the reference architecture of a VERSAG agent in the next sub-section.

2.3. VERSAG reference architecture

The conceptual architecture of a VERSAG agent is shown in Fig. 3. The core modules which make up a VERSAG agent are shown in a darker shade while two auxiliary services are shown in a lighter shade with dashed lines. We describe each of the modules below.

2.3.1. Platform specific agent

It is expected that VERSAG agents will be used to build applications on top of existing agent platforms such as JADE (Bellifemine et al., 2003) and Voyager (2007). Hence, each agent consists of a base, platform specific agent module which is the point of contact with the underlying agent runtime environment. Through it, basic services provided by the platform such as agent naming, communication and mobility are made available to the upper layers of the agent. It also provides the upper layers with methods to save and retrieve their state related data which needs to be maintained across agent migrations. These services are exposed through interfaces defined by the base agent so as to avoid any dependency on the underlying agent platform. While not within the scope of this study, by developing base agent implementations for different agent platforms, and cross-platform mobility mechanisms, it is possible to support VERSAG on multiple agent platforms with agents able to migrate between different platforms.

2.3.2. Kernel

The kernel is the agent's main controller. A VERSAG agent at its most basic level is itinerary driven and it is the agent kernel which implements this itinerant behaviour. The kernel's main responsibility is thus to retrieve itinerary commands from the itinerary service module and execute them. For this purpose, the kernel makes use of other modules and passes control to them when required. This behaviour of the kernel is implemented according to the "process coordinator" pattern (Gorton, 2006) as shown in Fig. 4.

An itinerary command consists of a primitive operation (one of *move*, *terminate*, *get*, *start*, *stop* and *discard*) that the kernel has to carry out. Thus, the kernel module implements support for executing an agent's primitive operations. The agent kernel cyclically executes itinerary commands that it retrieves from the itinerary service. Fig. 5 presents an algorithmic description of the kernel execution cycle.

2.3.3. Capability repository

The repository is the agent's personal storage space where its own and acquired capabilities are kept. It is part of the agent's data state and is carried along as the agent migrates. Capabilities held in the repository can be executed by the agent and transferred to other agents when requested (i.e. shared with peers). When an agent no longer requires a particular capability, or needs to migrate in lightweight mode over a slow link, it can discard capabilities from the repository.

2.3.4. Itinerary service

The itinerary service is a helper service to the kernel. It holds the agent's itinerary and provides methods to access itinerary commands and also to update the itinerary. Updating itineraries during execution is an important function as it is the approach in which an agent can change its actions during runtime. Another main purpose of this module is to hide the itinerary syntax and present other

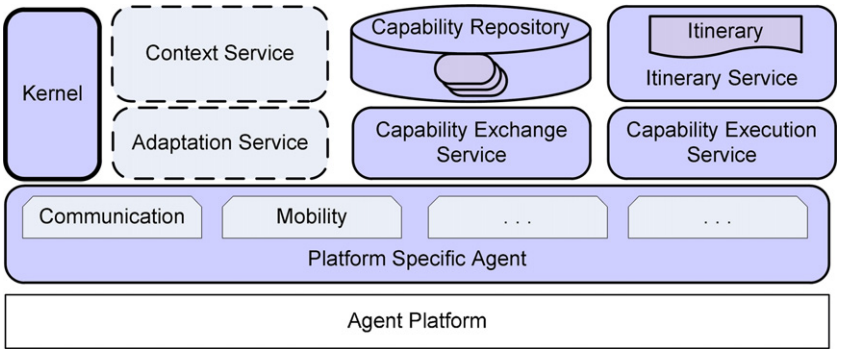

Fig. 3. Reference architecture of a VERSAG agent.

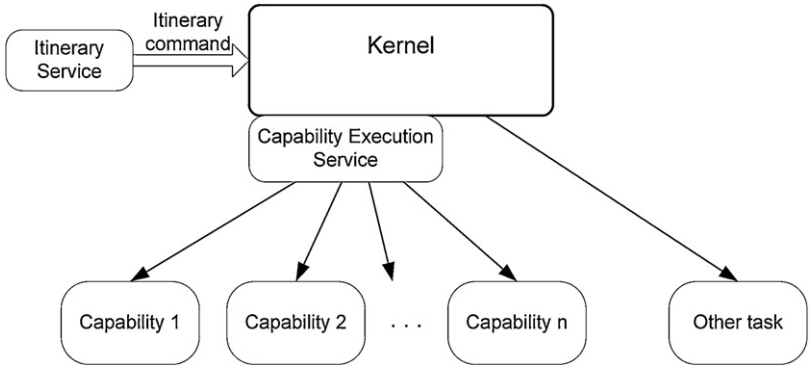

Fig. 4. The kernel implements the process coordinator pattern.

Algorithm: Kernel execution cycle
Input: *current* - itinerary command to execute, *prev* - previous itinerary command

```
1   Update status of prev
2   if current = ∅ or prev.status ≠ (EXECUTED or STARTED) then
        Block kernel (until woken up by prev command or an itinerary update)
    end if
3   if prev.status is FAILED_X then
        Call failure handling procedure
    end if
4   switch
        case current = MOVE:
            Make list of running capabilities to be restarted after move
            Request base agent to move
            Set current.status ← EXECUTING
            break
        case current = TERMINATE:
            Request base agent to terminate
            Set current.status ← EXECUTING
            break
        case current = DISCARD:
            if capability to unload in list of running capabilities then
                Stop capability
            end if
            Remove capability from repository
            Set current.status ← EXECUTED
            break
        case current = STOP:
            if capability to stop in list of running capabilities then
                Stop capability
            end if
            Set current.status ← EXECUTED
            break
        case current = START:
            Start capability
            Set current.status ← return value of Start
            break
        case current = GET:
            Get Capability Exchange service
            if Capability Exchange service unavailable then
                Set current.status ← FAILED_NO_REQUESTER_SERVICE
            else
                Invoke Capability Exchange service to find capability
                Set current.status ← EXECUTING
            end if
5   Set prev ← current
```

Fig. 5. VERSAG agent kernel execution algorithm.

modules (including capabilities) with a standard interface which remains unchanged in the event of changes to the itinerary syntax.

2.3.5. Capability execution service

The capability execution service provides the runtime environment for capabilities and is a key module of the agent. It enables starting and running multiple capabilities in parallel, providing each with its own execution space, and stopping them when required. The module is also responsible for providing communication mechanisms for capabilities. This module possesses its own execution thread and is called by the kernel when it is necessary to start or stop capabilities.

2.3.6. Capability exchange service

The important feature of peer capability sharing is provided by this module. It fulfills the dual roles of a capability requestor and provider. In its provider role, the module listens for capability requests from peer agents and responds as appropriate. The requestor role gives an agent the ability to request capabilities from peers. In most situations this service would run as a low priority process secondary to the agent's main tasks, and may be disabled when not required. The protocols used for capability exchange can be changed by replacing this module.

2.3.7. Auxiliary modules

In addition to the above, an agent can have many auxiliary modules. Auxiliary modules need to be implemented as VERSAG capabilities and two examples shown in Fig. 3 are an adaptation service and a context service.

The *adaptation service* contains the know-how to help an agent make its adaptation decisions. The adaptation process involves removing and acquiring capabilities, making changes to running capabilities and selecting suitable capabilities from multiple available ones. It takes input from the agent's itinerary, capability repository and context service to decide on the adaptation steps. This module incorporates multiple triggers for agent adaptation, leading to VERSAG's integrated view of agent adaptation. It is possible, in scenarios where agent adaptation is not required, for the module to be removed from an agent.

The *context service* is another auxiliary module which aids the execution of an agent. Since an agent is primarily a consumer of context information, this module consumes external context services to obtain contextual information about the agent's environment and also maintains internal agent-specific context information. For example, the context service can inform the agent of network parameters (e.g. bandwidth, latency, jitter) and device resource levels (CPU, memory, battery) to facilitate adaptation decisions/strategies. The module could be replaced with different implementations or removed when not needed.

An *itinerary generator* is another possible auxiliary service of an agent (not shown in Fig. 3). While a VERSAG itinerary is human-readable, it is not desirable to expose end users to a verbose and complex itinerary language as the method for interacting with agents. Thus, it is desirable to automate as much of the itinerary generation as possible in order to simplify the user's task of issuing requests to the agent. Preferably, users should be able to issue *declarative* requests to the agents without worrying about the step-by-step details of how it is to be carried out (e.g. similar to an SQL select statement). We envision that such a feature could be implemented in VERSAG via an itinerary generator capability. The capability takes a declarative user request as input, and produces a detailed itinerary for the agent as output. That is, an itinerary generator capability implements a function $f_{generate}: U \rightarrow I$ where U is a high-level declarative user request and I is an itinerary. Further investigation of itinerary generation is however beyond the scope of this study and is therefore not discussed in this article.

2.4. VERSAG capabilities

Capabilities are essentially reusable software components for which the necessary runtime environment is available within each agent (i.e. the capability execution service). Capabilities are important elements of the VERSAG framework as they provide agents with various application-specific behaviours and advanced features such as context-awareness and self-adaptability. Developing VERSAG based solutions involves designing and building new capabilities, rather than new agents as is the case normally with developing agent based systems. And, when new requirements arise, new capabilities can be developed and seamlessly introduced into the system. Since capabilities can be shared amongst agents, they also provide a fine-grained entity for migration and reuse in VERSAG based agent systems. It is through the dynamic acquisition, discarding and sharing of capabilities that VERSAG agents achieve their versatility.

A *capability model* defines how capabilities interact with other software elements and the runtime environment required to support their execution. A capability model is thus analogous to a component model as defined in (Councill and Heineman, 2001). We identify seven requirements and desirable features of a capability model for VERSAG.

- *Life cycle*. The capability model should define a standard mechanism for an agent to start and stop a capability without adversely affecting the agent's functioning or that of other independent capabilities executing on the agent. It should also be possible for an agent to execute multiple capabilities simultaneously.
- *Transportable/portable*. Capabilities should be packaged in a manner such that they are transportable over a computer network, allow agents to migrate while carrying them and to share them with other agents. They should also be portable across multiple platforms.
- *Interfaces*. Since application-specific capabilities would be built by third-party developers, a well-defined interface according to which they can be developed is essential. This interface should however be minimal, defining only what is required for the component to be executed by the underlying runtime environment. Capability developers would then have the opportunity to define custom interfaces which allow the capability to interact with other entities as needed.
- *Communication*. A capability should be able to communicate with other capabilities when needed and also access services provided by the agent. Support for such interactions should be enabled via the capability interfaces.
- *Naming and meta-data*. It should be possible to describe a capability with associated information such as a unique identifier, functionality contained within, input/output parameters, dependencies on other capabilities if any, hardware and software environment requirements and further descriptive meta-data.
- *Evolution support*. It may sometimes be necessary for different versions of a capability to co-exist in an agent, possibly for continuity during upgrading or to deal with conflicting dependencies (e.g. capability X and Y depend on versions 0.9 and 1.1 of capability Z respectively). Evolution support is therefore a desirable feature of a capability model.
- *Standards*. Standards allow capabilities to be reused in environments beyond VERSAG agents. Similarly, components from a wider environment would be available for use in VERSAG. Thus, a capability model underpinned by standards is a desirable feature.

During execution, capabilities can display different runtime behaviours. For example, while some capabilities would simply run

to completion, some may need to run continuously. To represent these differing runtime behaviours we group capabilities into three execution types as follows.

- *Oneshot.* A oneshot capability executes as a separate process (i.e. thread) inside the agent which runs to completion. For example, a capability which searches an SQL database for a particular record could be implemented as a oneshot capability. Typically, it is not necessary to explicitly stop a oneshot capability. The kernel is able to remove it from the list of active capabilities once the process terminates.
- *Cyclic.* A cyclic capability executes as a separate continuously running process inside the agent. For example, a context sensing capability which needs to continuously sense the environment could be implemented as a cyclic capability. It is necessary to explicitly request the agent to stop when such a capability needs to be terminated.
- *Passive.* A passive capability does not have a separate execution process. It only makes new functionality available for use by other modules. A passive capability, for example, could implement a data mining algorithm, but does not execute by itself. Instead, it makes its methods available for use by other modules.

A capability, as it is executed by an agent proceeds through several life cycle states. The first step is for the capability execution service to load a capability from the agent's repository and create an executable instance of it. Upon this initial creation, a capability is in a CREATED state. Then, as the capability starts running, it moves to a STARTED state for a cyclic or passive capability and to an EXECUTING state for a oneshot capability. A capability that completes successfully moves to an EXECUTED state. While a oneshot capability makes this transition without any external input, passive and cyclic capabilities need to be explicitly stopped through an itinerary command. If an error occurs at any stage of this process, the capability moves to a FAILED state. FAILED is a collection of states that are used to represent different causes of failures. These state transitions are illustrated in Fig. 6 with the many possible FAILED states grouped together for clarity. After a capability has been stopped or failed, it is removed from being an active entity inside the agent. However, it remains in the agent's repository as an inactive element and may be reused later.

2.5. Peer capability sharing

The second salient feature of VERSAG, as identified at the beginning of this section, is the ability of agents to share their functional capabilities with other agents in the environment. With this feature VERSAG agents avoid dependence on a centralized component source. This is advantageous in ubiquitous computing environments where dynamic variations can disrupt communication with the component source (e.g. due to network connectivity issues and possible source failure (Niemelä and Latvakoski, 2004; Spyrou et al., 2004)). Consider, for example, a scenario where a group of agents form an ad hoc network but are out of reach of the designated capability source. An agent in this network, which needs to acquire a capability, is unable to do so even if another agent in the network has the required capability since it can only be acquired from the designated source. An agent depending on a single component source can therefore experience performance degradation or task failure when it is unable to acquire required components. Furthermore, even when agents are able to communicate with the central source, the costs involved may be high. For example, if the communication path contains severely resource limited nodes, frequent and large data transfers may not be feasible. The ability of agents to request and acquire capabilities from multiple sources including nearby agents becomes highly valuable in such situations.

A VERSAG agent's GET primitive operation invokes the peer capability sharing process to search for and acquire a needed capability. The typical process consists of querying nearby agents for the needed capability(s), sorting the received responses to select a suitable capability/provider pair and requesting it from the relevant provider agent. Fig. 7 illustrates this sequence of steps in which an agent 'Bob' searches for and acquires a capability matching a given specification cs_x. A key issue associated with capability sharing is how an agent can make cost-efficient capability selections when multiple alternatives are available. In Gunasekera et al. (2010) we described and evaluated a multi-criteria cost model for making such decisions.

While peer capability sharing overcoming issues associated with a centralized component source, we recognize that it also introduces new concerns. The distributed and decentralized nature of peer capability sharing increases the complexity of managing capabilities. For example, it is necessary to ensure that capabilities do not disappear from the environment (e.g. due to all agents discarding their copies of a particular capability), keep track of

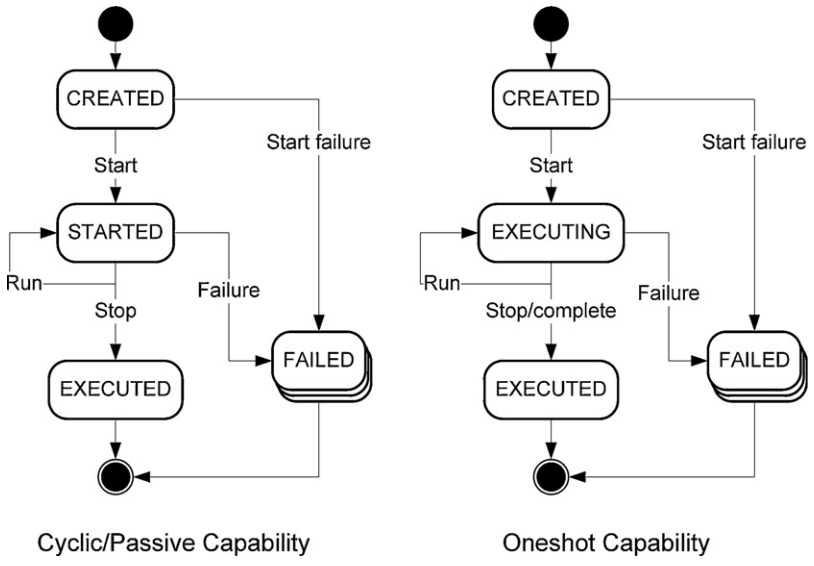

Fig. 6. State transitions for capabilities.

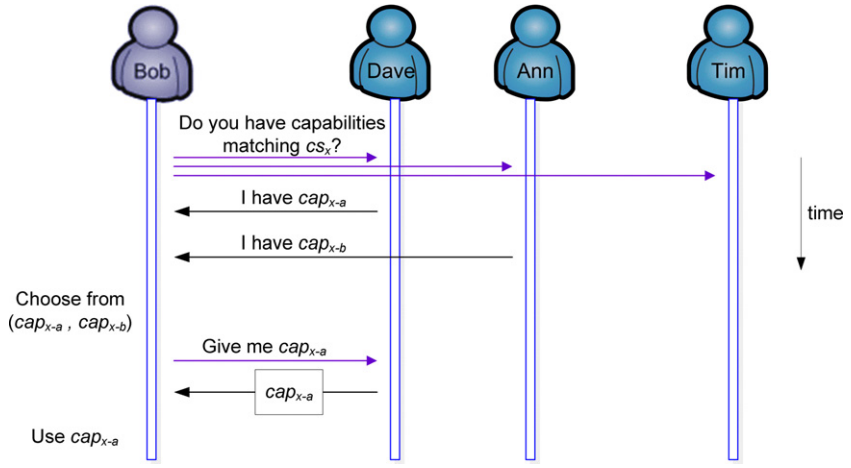

Fig. 7. Sequence of steps in capability search and acquisition without a directory service.

their usage and manage upgrades of capabilities. Discovering peer agents that are willing capability suppliers, searching through them and selecting suitable instances also incurs additional overheads on agent itinerary execution (in terms of time/network load/processing and so on).

Such concerns related to decentralization, performance and resource discovery are commonly identified within the peer-to-peer computing paradigm (Milojicic et al., 2002; Steinmetz and Wehrle, 2005) and have been addressed in a multitude of ways. Addressing these issues is not the focus of the current study. In Gunasekera et al. (2009b), as an extension of VERSAG, we proposed the use of agent teams rather than purely ad hoc sharing of capabilities in order to improve the performance of capability sharing agents. It should also be noted here that since VERSAG agents are expected to be deployed on top of an existing agent platform, it is possible to make use of infrastructure provided by the platform to solve some of these issues.

3. Design and implementation

We now proceed to describe our prototype implementation of the VERSAG framework. The prototype implementation was shaped by the following objectives and scope. The framework should be *application-agnostic* allowing it to be used to build applications in any suitable domain by incorporating necessary capabilities. It should also define a *capability model* that allows third-party developers to build capabilities that improve the core functions of agents as well as implement application-specific functions. The framework should be a lightweight layer on top of a current agent toolkit to avoid creating yet another agent toolkit (Carzaniga et al., 2007) and to enable reusing agent services provided by the underlying toolkit. Furthermore, this allows VERSAG agents to be introduced to existing multi-agent environments with minimum disruption. Finally, the prototype should be able to run on diverse hardware and software platforms, the likes of which are commonly encountered by ubiquitous computing applications.

3.1. Development platform and tool selection

The language of choice for mobile agent programmers, and consequently the platform used for developing most current agent toolkits (Alberola et al., 2010) is Java. JADE (Java Agent Development Framework) (Bellifemine et al., 2003) is one of a handful of currently active mobile agent toolkits, is compliant with FIPA (2011) standards for software agents and also provides support for the subsets of Java aimed at mobile devices (albeit with certain limitations in functionality due to device constraints). Therefore, JADE was chosen as the basis of our prototype. Programs written in Java are compiled to an intermediate bytecode format that is targeted towards a virtual machine rather than any specific hardware or operating platform. Consequently, these programs can be executed on any platform for which a compatible virtual machine implementation is available. Thus, the selection of Java and JADE is aligned with the objective of building a platform-independent prototype.

In Sub-section 2.4 we identified seven requirements of a VERSAG capability model. We now describe the choice of a capability model implementation for use with the prototype and justify its selection. The option of developing a capability model from ground up was not considered as the effort involved could not be justified considering there are already many candidates available. In addition, a custom developed capability model is unlikely to meet the requirement of being compatible with a widely used standard. We therefore investigated the possibility of reusing an existing component model for defining VERSAG capabilities.

The three possibilities we considered (Gunasekera et al., 2009c) are JADE, *μCode* and OSGi. We did not consider popular component models such as EJB (Enterprise Java Beans) and CORBA (Common Object Request Broker Architecture) (Emmerich and Kaveh, 2002) which target enterprise applications because they are too unwieldy for use inside a software agent and also not suitable for the types of resource constrained environments targeted by VERSAG.

- *JADE.*The JADE agent toolkit (Bellifemine et al., 2003) allows dynamic loading of functionality on to agents. A JADE agent's functionality is represented as programmer developed Behaviours (i.e. sub-classes of `jade.core.behaviours.Behaviour`) and adaptation support is provided with the aid of the `LoaderBehaviour` class and agent messaging. An agent that runs the `LoaderBehaviour` receives code for new Behaviours over ACL (Agent Commuication Language) messages and schedules them co-operatively with other Behaviours already in the agent. This approach results in the JADE agent behaviour life cycle and interface becoming that of the capability model. It thus results in the capability model becoming JADE specific and only allows limited control over the adaptation process. Furthermore, there is no built in meta-data support or support for evolution. Thus, JADE was not selected as the capability model implementation for VERSAG.
- *μCode.* μCode (Picco, 1998) is a flexible and lightweight Java toolkit for code mobility which could be used as the basis for building a capability model. A *group*, which can contain classes

and objects, is the unit of mobility in μCode. Code could be pushed to a destination or pulled by the destination as needed. It also allows multiple versions of the same class to co-exist within a single Java virtual machine, which could be the basis for building capability evolution support. It is possible to apply a standard for capabilities on top of μCode while other features required of a capability model have to be custom developed. It is no longer actively maintained and also requires considerable developments on top of it to meet our requirements. Hence, μCode too was not selected.

- *OSGi*. OSGi (2007) (originally the Open Services Gateway Initiative (Marples and Kriens, 2001)), known as the "dynamic module system for Java", is a standard component framework targeted at devices ranging from mobile phones and embedded devices to vehicles and high-end servers. It provides life cycle management of components, a well-defined contract for component development, supports co-existence of multiple versions of components and is an industry standard. OSGi components, named bundles, need to be developed in accordance with a simple API and are packaged as Java archive (JAR) files, making them transportable over networks. Bundles are named hierarchically as per java package naming conventions and a limited set of headers are supported by default, with the possibility of further extension by developers (OSGi, 2003). While heavier (in computational terms) than μCode, OSGi provides more features and is still lightweight as it supports small and embedded devices. In addition to a large population of available bundles (OSGi, 2007) (i.e. components), there are also multiple implementations of the OSGi framework (i.e. containers) available. Therefore, OSGi was selected as the basis for building the VERSAG capability model.

Once OSGi was selected as the basis for the VERSAG capability model, it was necessary to select a suitable OSGi implementation for use within the prototype. Five of the popular OSGi implementations currently available are as follows.

- *Apache Felix* (http://felix.apache.org/) is an open source implementation of the OSGi specifications and can be embedded within other applications.
- *Knopflerfish* (http://www.knopflerfish.org/) is an open source OSGi implementation with commercial support available.
- *Equinox* (http://eclipse.org/equinox/) is the reference implementation of the OSGi framework specification. It is also an open source project.
- *ProSyst* (http://www.prosyst.com/) provides commercial OSGi implementations that are used on devices ranging from mobile phones to vehicles.
- *Concierge* (http://concierge.sourceforge.net/) is an open source OSGi implementation that specifically targets resource constrained devices (Rellermeyer and Alonso, 2007).

Of the above, we limited our selection process to the four open source implementations. Apache Felix, Knopflerfish and Equinox target OSGi based applications on full powered computers (i.e. desktop and server computers) whereas Concierge was designed with resource constrained devices in mind. At less than 100 KB, the Concierge runtime also has a smaller file footprint than the other three open source implementations. It is however no longer actively maintained and only implements the OSGi specification release 3 (R3) whereas release 4.3 (R4.3) is the current version. This limitation notwithstanding, we decided to use Concierge for the capability model implementation of VERSAG as it provides the smallest and most resource friendly alternative. Furthermore, since OSGi R3 bundles are compatible with later releases, the possibility of later switching to a newer release remains open.

```
Properties props;
String name;
…
FrameworkV concierge = new FrameworkV();
//launch container in a new Thread
concierge.startContainer(props, name);
```

Fig. 8. Launching Concierge programmatically.

Several significant modifications needed to be carried out on the Concierge OSGi distribution in order to use it within VERSAG. The three main changes which were required are as follows.

1. *Enable embedding Concierge within an application/agent*. Since each VERSAG agent should have its own capability execution environment, each agent should have an OSGi container embedded within it. Concierge however does not come with support to embed it within another application. Therefore it was necessary to modify the Concierge source code to enable embedding. Starting a Concierge container is done by running the `ch.ethz.iks.concierge.framework.Framework` class. A new class `ch.ethz.iks.concierge.framework.FrameworkV` was developed to wrap around this with support to embed it within other applications. The code fragment in Fig. 8 shows how an OSGi container can be launched programmatically using this wrapper. The wrapper also contains methods to stop a container, install/uninstall bundles from the container, register external objects as OSGi services to make them accessible from within bundles, and to retrieve details of installed bundles and services as shown in Fig. 9.
2. *Remove dependency on configuration files at start up time*. When a Concierge container starts up, it requires certain configuration parameters, such as a list of bundles to start, locations of bundles, logging levels and buffer sizes to be provided to it. Concierge, like most software, was developed to be installed on and executed from a fixed location. Therefore it makes use of files on disk for reading initialization data. However, since we want to embed our OSGi container inside an agent that is mobile, the container cannot depend on files on disk for initialization (since it will be started at different computing devices with no access to any stored files). Therefore, we modified the Concierge source code to remove the dependency on configuration files and have all necessary parameters programmatically injected into it at start up time.
3. *Disable caching of bundles and temporary files on disk*. Similar to the use of configuration files, caching of bundles is useful only when the OSGi container is stationary and can rely on the disk being available as a long-term storage medium. This, clearly is not so in our situation. Furthermore, due to security and consistency needs, an agent migrating out of a device should not leave behind any residual information from its execution on the device's disk. Thus, we had to modify Concierge source code and disable the bundle caching.

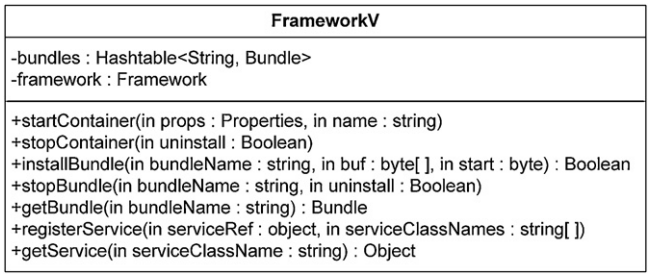

Fig. 9. Wrapper class to enable embedding Concierge.

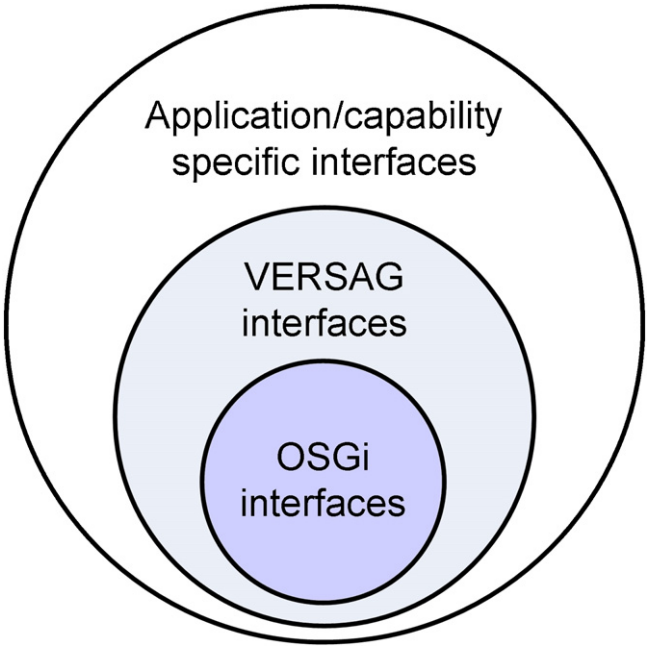

Fig. 10. Types of interfaces for VERSAG capabilities.

These three modifications were carried out on source code of the most recent release of Concierge OSGi (version 1.0.0RC3). While the modified container is no longer compliant with the OSGi specification, it still presents the same interface for bundles and is therefore compatible with standard OSGi bundles. Therefore, our bundle based capabilities still adhere to the OSGi standard. It is noteworthy that the modified version of Concierge still has a file footprint less than 100 KB which is an important requirement for VERSAG.

3.2. Interfaces for capability development

In Section 2.4, a set of well-defined and minimal interfaces according to which capabilities can be built was identified as a desirable feature of a VERSAG capability model. In our prototype implementation, three types of interfaces can be observed as illustrated in Fig. 10.

The minimum requirements of a capability are that it should be an OSGi release 3 (R3) (2003) compliant bundle supplemented with the meta-data required by VERSAG. An OSGi bundle consists of Java classes and other resources (e.g. icons, help files), which combine to provide some functionality, packaged as a Java ARchive (JAR) file. A bundle also has a manifest file (named MANIFEST.MF) that describes the bundle's contents and configuration information required to run it. A sample manifest file of a VERSAG capability is shown in Fig. 11. The OSGi specifications (2003) provide further details aiding development of bundles.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Description: Sample one shot bundle
Bundle-Name: oneshot1
Bundle-SymbolicName: versag.sample.oneshot1
Bundle-Version: 1.0.0
Bundle-Activator: simple.oneshot1.OneShot1Activator
Bundle-Vendor: VERSAG project
Import-Package: org.osgi.framework;version="1.3.0"
```

Fig. 11. Capability bundle manifest.

A capability is described with a unique identifier, function descriptions, list of supported platforms, execution type and various meta-data. In our prototype, these have to be separately specified as name-value pairs and associated with capabilities when they are loaded into VERSAG.

While the requirements for an OSGi bundle described above are the minimum requirements for a VERSAG capability, if a capability needs to interact with the agent and make use of its services, it needs to make use of the following interfaces and classes of a VERSAG agent.

- `mma2.AgentIF`
- `mma2.capability.Repository`
- `mma2.capability.Capability`
- `mma2.bundles.common.ServiceIF`
- `mma2.bundles.common.RequesterIF`
- `mma2.bundles.common.AbstractOneShotClass`
- `mma2.itinerary.ItineraryService`

These interfaces are required for tasks such as making use of agent services (`AgentIF`), retrieving/updating data stored in the agent's state (`Repository`) and modifying the agent itinerary (`ItineraryService`). Capabilities such as *context* and *adaptation* services which enhance agent autonomy are more likely to need such functionality as they need to make use of and manipulate agent internals while application-specific capabilities are less likely to need to use these interfaces.

In addition to the above two types of interfaces, capabilities can define their own interfaces in order to enable interactions between capabilities. VERSAG does not provide any guidelines on the granularity or functions implemented per capability. However, it is beneficial to build capabilities in a manner that promotes reuse. In order to facilitate this, we recommend a service-oriented approach with capabilities classified into two types as follows:

- *Service capabilities*. These implement generic functionality that can be reused in multiple domains and applications (e.g. a database driver capability, SQL client, statistical function library, Graphical User Interface library). These types of capabilities are developed by third parties and made available for use by application developers. For a given functionality, there would be multiple capabilities (potentially developed by different parties) that differ in terms of their features such as speed, system requirements, supported data types, license, price and so on. Thus, service capabilities are a form of commercial-off-the-shelf (COTS) components.
- *Coordinating capabilities*. The purpose of a coordinating capability is to join multiple service capabilities together in order to fulfil tasks that an agent has to carry out. For example, a coordinating capability implementing an information extraction function may combine different "document reader" capabilities in order to read different file formats on a computer. This type of capability has less scope for reuse and is expected to be developed by application developers who are aware of the different types of activities that the application may need.

Fig. 12 shows the involved capabilities and their dependencies for an information extraction capability set. The *File Processor* is the coordinating capability and requires appropriate *Reader* capabilities implementing the `ReaderIF` interface to do the actual reading of a data source. An appropriate reader is selected based on the type of file and invoked. Three readers for reading text and PDF files are shown. The *File Processor* then stores any extracted information in the agent's *Repository*. An independent *Result Handler* capability reads this information from the repository and handles it (e.g. display or do further processing).

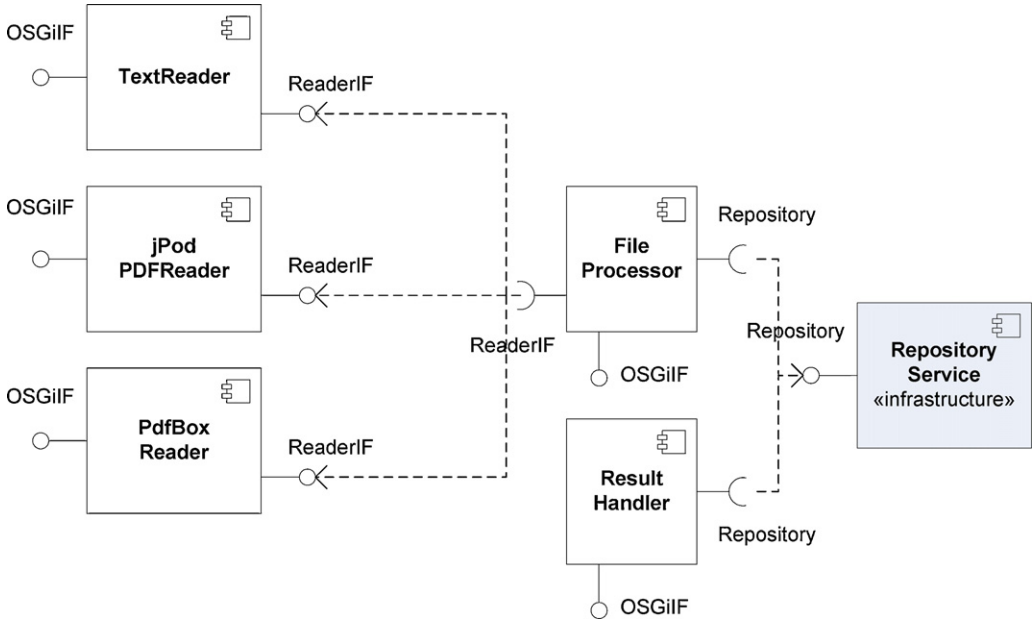

Fig. 12. Component diagram for information extraction capability set.

For this service-oriented approach to succeed, it is necessary to define interfaces that service capabilities adhere to and according to which coordinating capabilities can be programmed. This then allows different service implementations to be plugged-in based on capability availability and other needs (e.g. speed, environment requirements, and monetary limits). These are therefore application-specific interfaces that capability developers may need to adhere to. We note that while it is possible to have monolithic capabilities which combine these two types of functionality, they do not promote reusability and therefore should be limited for use in special circumstances.

3.3. Operational tools

We now provide a brief overview of the tools available for monitoring and managing a VERSAG based multi-agent platform. The JADE agent toolkit provides several tools for administration of agents/platforms and foremost amongst them is the graphical user interface agent (RMA agent/Remote Monitoring Agent). The RMA shows the agent platform, its containers (locations) and agents executing at each container in a tree view and allows creating/deleting/moving agents and sending ACL messages to agents. It is possible to launch other tools provided by JADE such as the Sniffer agent and Introspector agent from the RMA. These tools allow viewing agent internal details and monitoring communication amongst agents. Further details of these tools are available from the JADE Administrator's Guide (Bellifemine et al., 2007a).

As part of our study, we developed the *Controller Agent* (CA) and the *Remote Monitoring Console* (RMC) to control and monitor agent activity.

3.3.1. Controller Agent (CA)

The Controller Agent (CA) is a special (non-VERSAG) agent developed to conveniently control VERSAG agents through a command-line interface when running tests. While the RMA agent and associated JADE tools allow creating agents and sending messages via graphical interfaces, they can be cumbersome when a large number of agents have to be created and messaged to in a test environment. Furthermore, they cannot be used when a Windowing system is not available (e.g. on headless Linux machines and

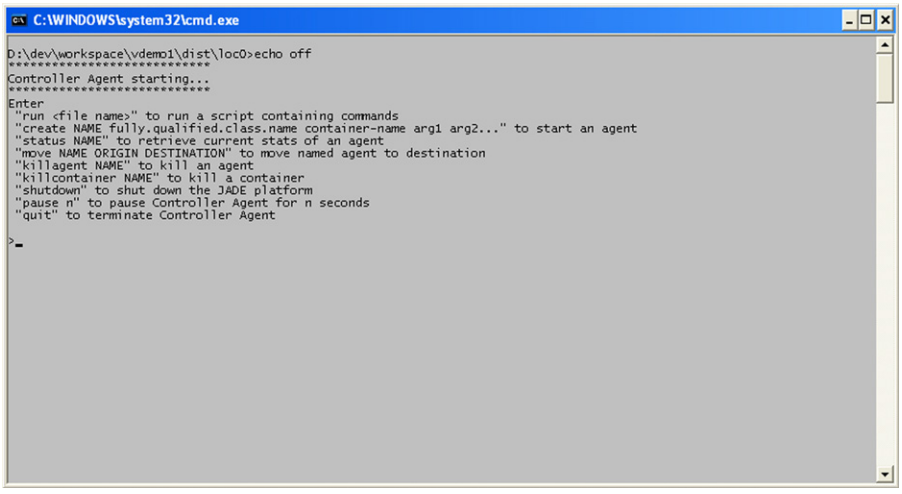

Fig. 13. The Controller Agent (CA) interface.

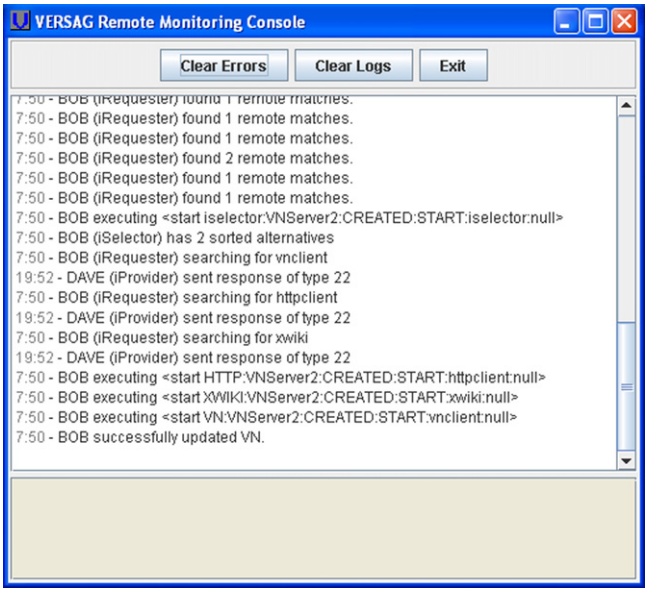

Fig. 14. VERSAG Remote Monitoring Console.

when remotely logging in using the SSH protocol). The CA provides a more convenient command-line and script file based approach to create, terminate and message agents. With the CA, agents can be controlled using simple text based commands, and multiple commands can be encapsulated in a script file that can then be executed through the CA's interface. The script file commands allow creating/moving/terminating agents, sending messages to agents, terminating a JADE container (i.e. a location), shutting down the agent platform, requesting agent status and pausing script execution for a desired number of seconds. A screen capture of the running CA listing its command syntax is shown in Fig. 13.

3.3.2. Remote Management Console (RMC)

The Remote Monitoring Console (RMC), shown in Fig. 14, is a graphical tool executed outside the agent system which displays specially formatted messages to provide a bird's eye view of the functioning of a VERSAG multi-agent system. JADE containers are configured to send selected log statements to the RMC over the network. Capability developers can incorporate appropriate log statements to describe their actions and any exceptional situations via the RMC.

4. Empirical evaluations

Having described the design and implementation details of our prototype, we now proceed to present empirical evaluations conducted to validate the usefulness of the VERSAG framework. We start this section with a justification for the selection of OSGi as the basis of a capability model and follow it up with a case study scenario which illustrates the functional benefits of VERSAG. The section ends with an experiment which compares the network efficiency of VERSAG agents with that of conventional mobile agents.

4.1. Performance justification for OSGi

Our selection of OSGi in Sub-section 3.1 raises an important issue of performance overheads. OSGi is targeted towards a wide range of devices and application domains and has many features required by them, which can be overheads when used in VERSAG.

To compare the possible overheads due to the use of an OSGi container instead of a custom developed capability model, a spike solution was built. The solution consisted of creating two simple capability models, one OSGi based and one custom built, on top of JADE agents. Two sample capabilities (*A* and *B*) were also developed for testing. The number of source statements, class sizes and average capability execution time were measured. The results of the performance comparison are shown in Table 1.

Table 1
Comparison of custom-developed Vs OSGi based capabilities.

Capability	Average execution time (ms)	Source code statements	Class size (KB)
A – Custom version	34	46	1.75
A – OSGi version	35	86	4.56
B – Custom version	287	43	1.73
B – OSGi version	286	83	4.55

The results show that capability execution times are similar for both the OSGi version and the custom version. This is important since it indicates that the extra functionality of OSGi does not translate into slower execution of custom loaded classes.

In terms of size, we observe that the OSGi source code has more source statements and that its binary version (i.e. JAR file) too is larger. This is due to the additional OSGi wiring code required. For example, an OSGi bundle needs to have a Bundle Activator class that implements the `org.osgi.framework.BundleActivator` interface. This class contains methods which are invoked by the container in order to start and stop the bundle. This wiring code however is fixed (40 source statements in the examples above) and will be an insignificant contributor to the capability's size as the size of the functional logic increases. The class size follows the trend of source statements and is unlikely to be significant.

Furthermore, since mobile devices are one class of devices targeted by OSGi, resource constraints of such environments have been taken into account in its design. The modular nature of the OSGi specification, where additional services themselves are implemented as OSGi bundles that can be added and removed from the container assists our objective of keeping the OSGi container as lightweight as possible. Thus, we can conclude that the use of OSGi as the capability model implementation does not compromise our objective of building a lightweight agent framework.

4.2. Case study

A fundamental advantage VERSAG agents have over conventional agents is the ability to acquire new behaviours for situations which are not anticipated at design time. As a result of this feature, a VERSAG agent is able to continue execution in situations where a conventional agent would have to be replaced. This ability makes them especially suitable for use in dynamic and heterogeneous environments. We now describe a case study scenario in order to demonstrate and establish this functional benefit of VERSAG agents. Specifically, we demonstrate the ability of an agent to dynamically acquire new capabilities and use them to fulfil an allocated itinerary. The ability of an agent to search for and acquire capabilities it needs from other agents and the ability to make a cost-efficient selection when multiple suitable capabilities are available is also demonstrated.

Our case study builds on the motivating scenario from Sub-section 2.1. A human user Bob uses his virtual personal assistant to automate some office work on a typical workday morning. While on his way to work by train, Bob types a memo, which needs to be uploaded to the office virtual noticeboard, on his mobile computing device. Once he arrives at the office, Bob instructs his trusted virtual personal assistant agent (also named "Bob") to upload this memo to the noticeboard. The agent is also assigned a second task: to search two file servers in the office Intranet for documents which refer to the phrase "mzone kiosk". Bob indicates that he

Table 2
Locations of the case study agent environment.

Location	Description
VNServer2	Virtual noticeboard access device
Server5	File server 1
Server6	File server 2
BobLap	Agent *Bob* starts and ends his itinerary here
DavePC	Agent *Dave* resides here
AnnPC	Agent *Ann* resides here
TimPC	Agent *Tim* resides here
loc0	Controller Agent deployed here
loc1	Main container of distributed JADE platform

would like to access this list of documents from his laptop computer.

We use a laptop computer to represent Bob's mobile computing device on which the agent initially resides. This is due to the limited support for deploying mobile agents on current devices such as mobile phones and personal digital assistants (PDA). The laptop computer is connected to the rest of the network over an IEEE 802.11g wireless network and is resource-restricted in comparison to desktop computers. The virtual noticeboard is represented by an XWiki (http://www.xwiki.org) page named "NoticeBoard". XWiki was chosen since it allows REST style (Richardson and Ruby, 2007) client interactions and therefore is representative of a large group of similar resources that an agent may have to interact with. We use a distributed JADE agent platform with nine locations (i.e. JADE containers (Bellifemine et al., 2007b)) for the experiment. Table 2 lists these locations and their purposes. A graphical representation of the environment, agents and the migration path to be taken by agent Bob is provided in Fig. 15. In this setup, agents Dave, Ann and Tim are the peer agents with whom agent Bob may share capabilities.

4.2.1. Upload memo to office virtual noticeboard

We instruct agent Bob (via an itinerary sent using the CA) to carry out the first task assigned by Bob, which is to migrate to the office network and upload the typed memo to the virtual noticeboard.

The RMC log messages shown in Fig. 16 indicate the steps in carrying out this task. (We note that differences in time and time format are due to the messages coming from computers with different system times and time formats.) Lines 4 and 5 of the logs show that agent Bob unloads the *editor* capability and migrates to *VNServer2*. After arrival at *VNServer2*, Bob queries its peers for the necessary capabilities, namely *httpclient*, *xwiki* and *vnclient* as shown in lines 11 and 12. The three peer agents Dave, Ann and Tim all respond to the query (lines 13–15) with responses of "type 21" containing details of matching capabilities they possess. Agent Bob then deliberates on these responses and selects a capability provider (agent *Dave* in this case) as shown in lines 22 and 23. Lines 24–29 show the capabilities being acquired (A response of "type 22" contains a capability). Finally, the capabilities are executed in sequence to complete the task. A screen capture of the messages as seen on the RMC is shown in Fig. 14. The updated XWiki page with Bob's memo is shown in Fig. 17.

This step illustrates the ability of a VERSAG agent to dynamically search for and acquire new capabilities from other agents within the environment and use them to fulfil its assigned itinerary. When agent Bob was assigned the itinerary at the beginning of this step, it did not contain the necessary know-how to update the office virtual noticeboard. Thus, this is an itinerary that a conventional mobile

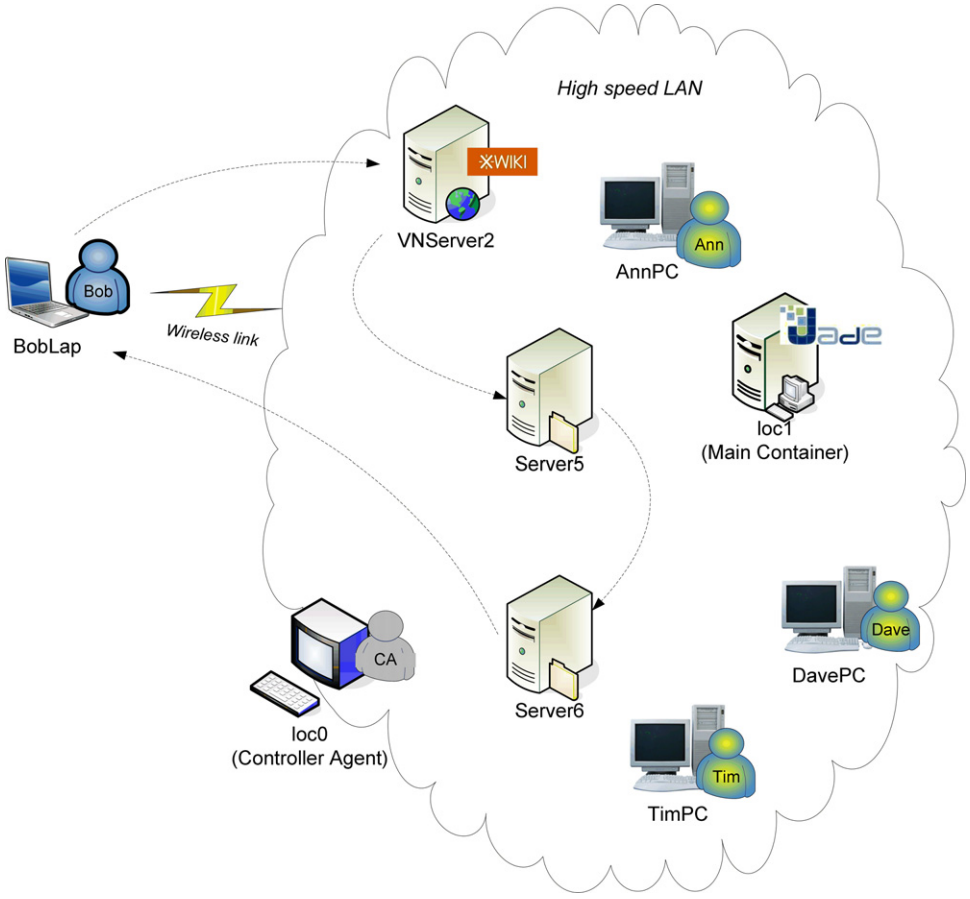

Fig. 15. Overview of case study agent environment.

```
1   19:48 - BOB executing <start editor:BobLap:CREATED:START:editor:null>
2   19:50 - BOB Editor accepted user content.
3   19:50 - BOB received a new itinerary
4   19:50 - BOB executing <unload editor:BobLap:CREATED:UNLOAD:editor:null>
5   19:50 - BOB executing <move VNServer2:BobLap:CREATED:MOVE:null:VNServer2>
6   19:50 - BOB is moving elsewhere
7   7:50 - BOB arrived at VNServer2
8   7:50 - BOB executing <start contextapi:VNServer2:CREATED:START:contextapi:null>
9   7:50 - BOB executing <start ahpcostmodel:VNServer2:CREATED:START:ahpcostmodel:null>
10  7:50 - BOB received a new itinerary
11  7:50 - BOB executing <find spec const;TIME:5;LOAD:4 HTTP;http_client_methods;JADE3_5
    XWIKI;xwiki_client_methods;JADE3_5
    VN;virtual_noticeboard_client_aggregator;JADE3_5:VNServer2:CREATED:FIND:null:null>
12  7:50 - BOB (iRequester) searching by spec
13  19:52 - DAVE (iProvider) sent response of type 21
14  7:50 - ANN (iProvider) sent response of type 21
15  19:52 - TIM (iProvider) sent response of type 21
16  7:50 - BOB (iRequester) found 1 remote matches.
17  7:50 - BOB (iRequester) found 1 remote matches.
18  7:50 - BOB (iRequester) found 1 remote matches.
19  7:50 - BOB (iRequester) found 2 remote matches.
20  7:50 - BOB (iRequester) found 1 remote matches.
21  7:50 - BOB (iRequester) found 1 remote matches.
22  7:50 - BOB executing <start iselector:VNServer2:CREATED:START:iselector:null>
23  7:50 - BOB (iSelector) has 2 sorted alternatives
24  7:50 - BOB (iRequester) searching for vnclient
25  19:52 - DAVE (iProvider) sent response of type 22
26  7:50 - BOB (iRequester) searching for httpclient
27  19:52 - DAVE (iProvider) sent response of type 22
28  7:50 - BOB (iRequester) searching for xwiki
29  19:52 - DAVE (iProvider) sent response of type 22
30  7:50 - BOB executing <start HTTP:VNServer2:CREATED:START:httpclient:null>
31  7:50 - BOB executing <start XWIKI:VNServer2:CREATED:START:xwiki:null>
32  7:50 - BOB executing <start VN:VNServer2:CREATED:START:vnclient:null>
33  7:50 - BOB successfully updated VN.
```

Fig. 16. Complete RMC log message.

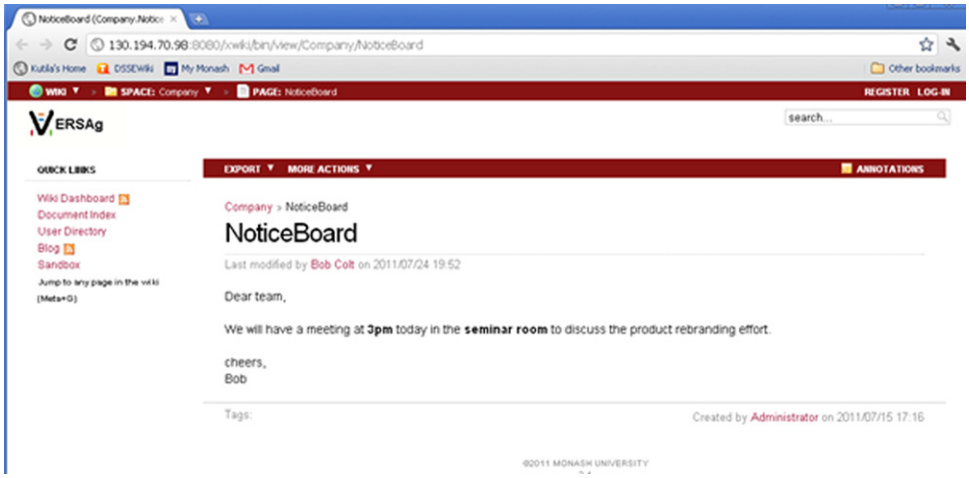

Fig. 17. Bob's memo on the virtual noticeboard.

agent would not be able to fulfil. The peer capability sharing feature allowed Bob to first search nearby agents for the necessary know-how (i.e. capabilities which implement the relevant functionality), acquire these capabilities and execute them in order to fulfil the itinerary.

4.2.2. Search and locate documents containing specific search query

The second task assigned to agent Bob is to search through documents in two Intranet file servers and identify documents which contain the phrase "*mzone kiosk*". Once again agent Bob does not have the required capabilities and needs to acquire them from peer agents. A tabular format of the itinerary to follow is shown in Table 3. The *get* operations encapsulate VERSAG's cost-efficient capability acquisition process which in this instance takes network load and time as relevant cost criteria.

Table 3
Itinerary to search documents.

Location	Operations
VNServer2	discard *vnclient xwiki httpclient*, move *Server5*
Server5	get [*2common pdfreader txtreader 2matcher*] start *2common*, start *pdfreader*, start *txtreader*, start *2matcher* move *Server6*
Server6	start *2matcher*, move *BobLap*
BobLap	get [*resultgui*], start *resultgui*

Agent Bob searches for the relevant capabilities and finds multiple instances available from the peer agents. Agent Dave has two *pdfreader* capabilities, namely: *2pdfboxreader* and *2jpodpdfreader*. Also, the *txtreader* capability is available with agents Dave and Ann. This leads to four possible combinations of capabilities (i.e. capability groups). A multi-criteria decision making cost model

```
24/07/11 20:35 [21] FINE bundle.costmodel.ahp1.AhpCostModel  - After sorting:
2matcher[130.194.70.98:1230] 2jpodpdfreader[130.194.70.98:1230]
2txtreader[130.194.70.98:1230] 2common[130.194.70.98:1230] ) = 0.3531391
2matcher[130.194.70.98:1230] 2jpodpdfreader[130.194.70.98:1230]
2txtreader[130.194.70.183:1230] 2common[130.194.70.98:1230] ) = 0.35288256
2matcher[130.194.70.98:1230] 2pdfboxreader[130.194.70.98:1230] 2txtreader[130.194.70.98:1230]
2common[130.194.70.98:1230] ) = 0.1470322
2matcher[130.194.70.98:1230] 2pdfboxreader[130.194.70.98:1230]
2txtreader[130.194.70.183:1230] 2common[130.194.70.98:1230] ) = 0.14694609
```

Fig. 18. Agent Bob's logs showing utilities of sorted capability groups.

(Gunasekera et al., 2010) is used by agent Bob to select the most cost-efficient capability group. The sorted groups and their corresponding utility values as logged by agent Bob are shown in Fig. 18.

It could be observed that the *2jpodpdfreader*, which is roughly one fourth the size of the *2pdfboxreader* (i.e. 824 and 3176 KB respectively), was selected by agent Bob. Since no network traffic is generated during execution of these two capabilities, network load is only due to capability acquisition. Therefore, the smaller size of the *2jpodpdfreader* puts it at a significant advantage. Considering time estimates also, according to capability meta-data *2jpodpdfreader* is deemed the faster of the two. Thus, we see that the agent has selected the more cost-efficient alternative. This therefore illustrates the ability of a VERSAG agent to make a cost-efficient selection based on contextual input and user preferences.

Agent Bob migrates to *Server6* carrying these acquired capabilities and reuses them there for searching the file system. These capabilities are then discarded prior to migrating to the laptop (*BobLap*) to display results. There again agent Bob searches for a suitable capability to display results. A screen capture of the result displaying GUI capability presented in Fig. 19 shows the phrase that was searched for, files containing the search phrase and the word count of each file.

This scenario demonstrated the ability of VERSAG agents to locate nearby agents and share capabilities with them when required. It also showed that when several alternative capabilities are available, a VERSAG agent has the ability to select the most cost-efficient alternative to fulfil its itinerary, taking into account user preferences, capability details and contextual inputs. It is this versatility of VERSAG agents that make them suitable in uncertain situations where conventional mobile agents cannot be applied.

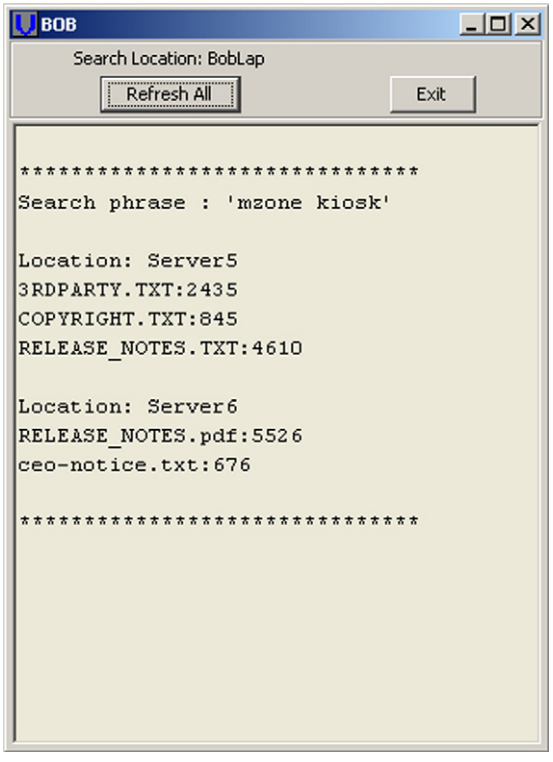

Fig. 19. Agent displaying search results to Bob on the laptop.

4.3. Efficient migration through capability sharing

A key advantage of VERSAG is the ability of an agent to acquire and discard capabilities based on needs, which allows it to stay lightweight despite the diverse behaviours displayed and heterogeneous conditions encountered during its lifetime. This ability can lead to reductions in overall network traffic generated by migrating agents. We now describe a set of experiments which illustrate this behaviour. Specifically, our objective is to demonstrate that a VERSAG agent performing a task which requires it to carry out distinct sub-tasks at different locations generates less network traffic compared to a conventional mobile agent assigned with the same task (Gunasekera et al., 2009a).

4.3.1. Experimental setting

Let us consider that Bob's personal assistant agent from the case study scenario has to migrate to computers in the office Intranet searching different document sources for specific content. We assume that the agent has identified a sequential itinerary (migration path) which takes it through n locations (i.e. computers). At each location, the agent has to search a different type of document source. For example, it has to search a MySQL database at one location, a collection of PDF files at another location, an LDAP directory at another and so on. The final task is for the agent to move to Bob's laptop computer and display aggregated search results. Since the tasks are distinct, the agent requires a different capability for each task. Even though the agent does not have any of the required capabilities up front, they are available with other agents in the Intranet and can be acquired from them.

Fig. 20 illustrates our test setup for the above scenario. The number of locations is n and the agent's starting and terminating point is identified as location N_0. At each intermediate location N_i, where $0 \leq i \leq n$, the agent has to perform $task_i$ which is only required at that location. The final task, $task_n$ is performed at location N_0 after having traversed all locations. We denote migration from location N_{i-1} to N_i as mig_i. The final migration from N_{n-1} to N_0 is specified as mig_n.

We use two types of agents to fulfil this task as described below.

Case I: The itinerary is assigned to a conventional mobile agent with all the functions required to execute the various tasks coded into it

Case II: The itinerary is assigned to a VERSAG agent with a "light-travel" policy. That is, at location N_i the agent acquires capabilities necessary to perform $task_i$, uses them and discards them before migrating to the next location. Hence, the agent always migrates as an empty agent without carrying any capabilities

For Case II, functions required for $task_i$ are implemented as a single capability. In the experimental setup, a separate agent

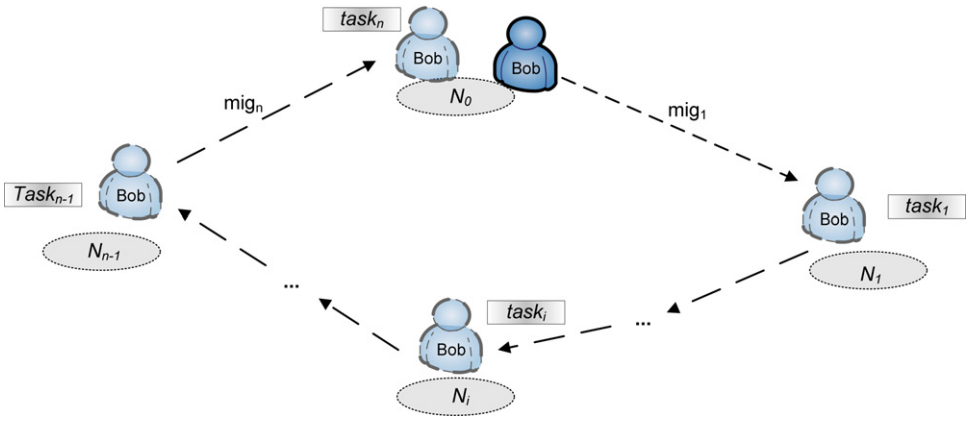

Fig. 20. Agent migrates to n locations, performing a distinct task at each location.

residing at location N_0 contains all these capabilities and supplies them to our working agent when requested. It is assumed that the task results which are carried by the agent are negligible in size. The task at each location consists of searching through a collection of files (located on the file system at that location) for a given phrase and then recording the name and word count of the files which contain the phrase. At the final location, $task_n$ is to display the collected results on the computer console. As stated earlier, different types of files have to be read at each location and the agent therefore requires a different capability at each location. For Case I, the conventional mobile agent is built with the necessary functions to search all the types of files and display results.

The JADE agent platform implements a *pull-per-class* migration strategy where the destination container (i.e. location) pulls classes from the agent's home container as and when required (Braun et al., 2005). As a consequence, only classes that are required at a particular location are transferred. Furthermore, due to the higher number of network connections, migration overheads are higher in comparison to mechanisms where a collection of classes are moved together as a single unit. To negate the effects of these JADE specific behaviours, we instantiate all custom developed classes in the agent's main class to ensure that they are always moved. In addition, in the conventional mobile agent (i.e. Case I), third-party libraries required by it are artificially carried as agent data.

The experiments are carried out in a high-speed Local Area Network (LAN), using computers running Windows and Solaris operating systems and Java SDK 1.6.0. The agents are implemented using JADE version 3.5.

The number of locations n, is varied from 2 to 10. The total number of tasks, and therefore capabilities, for a test run is also equal to n. The number of file types supported by the conventional agent increases with n, making the agent larger as n increases. In each test run, a conventional mobile agent (Case I) and a VERSAG agent (Case II) are allowed to complete the itinerary. The agent platform is restarted between test runs to ensure that any code caching mechanisms do not affect the measurements. Total network traffic generated and time taken to complete the itinerary are measured.

4.3.2. Results and analysis

A graphical representation of mean network load variation with number of locations is provided in Fig. 21. The results clearly indicate that VERSAG agents generate less network load in comparison to a conventional mobile agent as the number of locations traversed increases.

To aid further analysis and interpretation of the experiment results, we build an analytical model of network load for the two

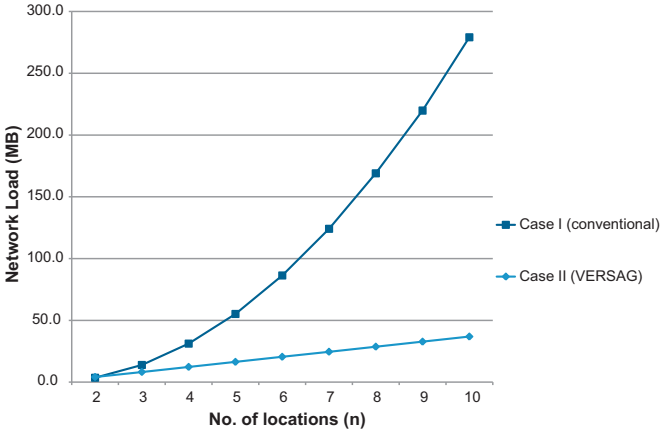

Fig. 21. Network load Vs number of locations.

cases. We assume that overheads for migration and capability exchange (B^{ovrhd}_{mig} and $B^{ovrhd}_{c\ ex}$) are constant values and that the size of the accumulated result carried along with the agent is negligible. We also assume that the total size of classes implementing the tasks in the conventional mobile agent is equal to the sum of capability sizes in the VERSAG agent.

4.3.2.1. Case I. Network traffic generated due to agent migration mig_i can be represented as the sum of the agent's class size (B^C_{MA}), size of the serialized agent (B^{obj}_{MA}) and migration overheads (B^{ovrhd}_{mig}). When the agent migrates to its home location (N_0), there is no need to send the agent classes since they are already available at the destination.

$$B_{mig_i} = \begin{cases} B^C_{MA} + B^{obj}_{MA} + B^{ovrhd}_{mig} & \text{if } i < n \\ B^{obj}_{MA} + B^{ovrhd}_{mig} & \text{if } i = n \end{cases} \quad (1)$$

We further breakdown the agent's classes as those of the base agent (B^c_{ma}) and those of the various capabilities ($B^C_{\sum cap_i}$). That is,

$$B^C_{MA} = B^C_{ma+\sum cap_i} \quad (2)$$

Total network load for a conventional mobile agent can be represented as follows.

$$B_I = B_{mig_1} + B_{mig_2} + \ldots + B_{mig_n}$$

Expanding this with Equations (1) and (2),

$$B_I = (n-1)\left(B^C_{ma+\sum cap_i} + B^{obj}_{ma+\sum cap_i} + B^{ovrhd}_{mig}\right) + \left(B^{obj}_{ma+\sum cap_i} + B^{ovrhd}_{mig}\right)$$
$$B_I = (n-1)B^C_{ma+\sum cap_i} + n\left(B^{obj}_{ma+\sum cap_i} + B^{ovrhd}_{mig}\right) \quad (3)$$

4.3.2.2. Case II. Total network traffic for the VERSAG agent includes the cost of n migrations and n capability exchanges.

$$B_{II} = \left(B_{mig_1} + B_{mig_2} + \ldots + B_{mig_n}\right) + \left(B_{c-ex_1} + B_{c-ex_2} + \ldots + B_{c-ex_n}\right) \quad (4)$$

Network traffic due to a capability exchange consists of capability class size ($B^C_{cap_i}$) and the exchange overhead (B^{ovrhd}_{c-ex}), or is zero when capability provider and requester are co-located (i.e. at location N_0). Here the exchange overhead includes capability request response message sizes.

$$B_{c-ex_i} = \begin{cases} B^C_{cap_i} + B^{ovrhd}_{c-ex} & \text{where } i = 1 \ldots (n-1) \\ 0 & \text{where } i = n \end{cases} \quad (5)$$

With B_v representing size of a VERSAG base agent, total network load is:

$$B_{II} = (n-1)\left(B^C_v + B^{obj}_v + B^{ovrhd}_{mig}\right) + \left(B^{obj}_v + B^{ovrhd}_{mig}\right) + \left((B^C_{cap_1} + B^{ovrhd}_{c-ex}) + \ldots + (B^C_{cap_{n-1}} + B^{ovrhd}_{c-ex})\right)$$
$$B_{II} = (n-1)B^C_v + n\left(B^{obj}_v + B^{ovrhd}_{mig}\right) + B^C_{\sum cap_1} + (n-1)B^{ovrhd}_{c-ex} \quad (6)$$

Equations (3) and (6) enable us to understand the trends displayed by the two plots in Fig. 21. The plot for Case I (conventional agent) displays a quadratic increase where as the plot for Case II (VERSAG agent) only increases linearly. In the experiment, the base agent class/object sizes and overheads are constants while the number of locations (n), class size of all capabilities ($B^C_{\sum cap_i}$) and object size of all capabilities ($B^{obj}_{\sum cap_i}$) are variable. Thus, we observe that Equation (3) for a conventional agent contains products of the variable terms leading to a quadratic plot while Equation (6) for a VERSAG agent only contains the variable terms in isolation leading to a linear increase. Furthermore, we note that for VERSAG to produce lower network load than a conventional mobile agent, we should have $B_{II} \leq B_I$.

While we do not formally analyse time consumed for itinerary completion, our experimental data shows that the conventional agent consumes more time as the number of locations increases. We note though that time taken by VERSAG agents would increase as more complex protocols are used for capability discovery and acquisition.

The results from our empirical and analytical evaluations clearly demonstrate and validate the performance gains from using a VERSAG agent to carry out distinct sub-tasks at a set of distinct locations as opposed to a conventional mobile agent assigned with the same task.

If we relax the requirement of a distinct capability at each location and consider that there are m capabilities which could possibly be used over n locations, a conventional agent would still need all m capabilities embedded in it. If the VERSAG agent in this situation continues to discard each capability after use, it ends up requesting the same capability multiple times and may end up being less efficient than the conventional agent. Instead, the VERSAG agent could further improve its efficiency by using more sophisticated capability management mechanisms which take into account criteria such as the probability that a capability would be required again, cost of links to be traversed, ease of reacquiring the capability at another location and throughput requirements of the application. In (Gunasekera et al., 2010), we proposed a cost model which allows agents to make cost-efficient capability selection decisions. This cost model was used by the agents in the case study of Sub-section 4.2 for their decision making.

5. Related work

There exist a number of related projects where mobile agent technology has been used to build software for ubiquitous computing environments. MobiSoft (Erfurth et al., 2008) is one such project which investigates the use of mobile agents in the role of personal assistants to human users. They use a custom-built Java like language named TAL (The Agent Language) to aid agent migration on J2ME mobile devices. MobiSoft agents are however not functionally adaptive and therefore have fixed functionality unlike VERSAG agents. ODDUGI (Choi et al., 2009) is a Java based mobile agent system which emphasizes reliability, security and fault tolerance as important features in ubiquitous computing environments. AFME (Agent Factory Micro Edition) (Muldoon et al., 2007) is a BDI mobile agent framework targeting resource constrained J2ME mobile devices. A survey and classification of mobile agents in ubiquitous computing are presented in (Zhang et al., 2012). While there is a considerable amount of literature on the use of mobile agent technology in ubiquitous computing environments, research where mobile agents exhibit ability to dynamically adapt their internal structures, as seen in VERSAG, is extremely rare in the literature. Below, we briefly introduce the important works in this area.

Dynamically Configurable Software (DynamiCS) (Tu et al., 1998) pioneered the idea of mobile agents being containers for application-specific plug-in components. A DynamiCS agent by default contains basic mobility and persistence capabilities while most application semantics are implemented as plug-in components that can be loaded and unloaded from an agent at runtime. It was targeted towards ecommerce applications and specifically limited itself to adapting agents to support different negotiation schemes. The research on negotiating agents by Parakh et al. (2002) shares similar objectives to DynamiCS but is implemented using different base technologies.

The *dynamic agent* infrastructure (Chen et al., 1999) is another pioneering effort which provides agents with dynamic-modifiability of behaviours. A dynamic agent is composed of a fixed carrier part that provides housekeeping services and a dynamic part which contains useful capabilities. An agent presented with a task that requires capabilities beyond what it has at present can dynamically load new programs from a specified remote location. Thus, their adaptation is for functional requirements and has to be explicitly specified in the task assigned to the agent.

The work on dynamically adaptable mobile agents (DAMA) (Brandt, 2001) investigates how mobile agents can dynamically

adapt their functionality to suit different computational environments they encounter. A DAMA mobile agent is made up of a small non-adaptable (environment independent) core, adaptation framework and adaptable parts that depend on the environment. Adaptation in DAMA agents therefore occurs only as a result of agent migration to a different environment.

The Generic Adaptive Mobile Agent architecture (GAMA) (Amara-Hachmi and Fallah-Seghrouchni, 2005) is another such project where agents adapt to suit new environments. GAMA targets ubiquitous computing environments and an agent in GAMA is composed of multiple components and adapts itself after migration to suit its new environment. In adaptation, components can only be selected from a collection available at the agent's destination.

Roam (Chu et al., 2004) is a framework where applications can be dynamically synthesized using agent-like migratory entities termed *Roamlets*. The Port-Based Adaptable Agent Architecture (PB3A) by Dixon et al. (2000) has a port-based module (PBM) as its core building block and has a port-based agent (PBA) as an autonomous entity. The Self-Configuring Personal Agent (SCPA) platform (Feng et al., 2008) is a recent effort targeted towards creating personal assistant agents in ubiquitous computing environments. These agents can adapt in response to application requirements as well as environmental changes.

VERSAG improves the state-of-the-art in several directions. A key weakness observed in existing literature is the dependence of an agent on a single component source. DAMA is the only exception to this with its use of proxy repositories in an attempt to overcome this limitation. In VERSAG, we eliminate dependence on a single component source through the concept of sharing functional capabilities among peer agents. This not only increases the utility and efficiency of mobile agents but is also of significant value in ubiquitous computing environments where network connectivity issues and high communication costs can make dependence on a centralized component source undesirable. Another weakness observed in existing research is their fragmented view of adaptation, where adaptation is necessitated by either having to fulfil application tasks for which the required know-how (i.e. functionality) is not available with the agent, having to meet the requirements of a new device that the agent is migrating to, or as a result of environmental changes (sensed by itself or another context-aware agent). However, all these are valid needs for adaptation and it is necessary for agents in ubiquitous computing environments to possess an integrated view which takes into account all possible adaptation triggers. With VERSAG, we present a holistic view of adaptation allowing agents to adapt in response to any of these different triggers. A third improvement over the state-of-the-art is increased reusability of components in VERSAG. In previous research, agent components can only be reused within that particular framework. By adopting the widely accepted OSGi specification as the basis of our capability model, VERSAG expands the scope of reuse for its capabilities to the many other situations where OSGi is used.

6. Discussion

While VERSAG was designed primarily as a framework for building ubiquitous computing applications, its applicability extends to other domains where software agents are deemed useful. For example, in (Gunasekera et al., 2011) we described how VERSAG agents can improve the efficiency of service-oriented software agents. In many agent based systems, agents are not migratory and their environments are not severely resource-restricted as in ubiquitous computing. Therefore, we briefly introduce two scenarios to illustrate how VERSAG brings significant values in such situations.

Let us first consider a long-lived software agent designed to carry out a fixed set of tasks and residing in a non-resource constrained environment. Tasks allocated to such agents are generally complex (e.g. negotiation and purchase in electronic auctions, control and monitoring of industrial equipment). Over time, better implementations of these tasks appear making the agents inefficient and sometimes obsolete. For example, a negotiating agent would be at a significant disadvantage if rival agents use improved negotiation strategies. In such a situation, the agent needs to be replaced with another agent that has the improved strategies. Similarly, a software agent that controls and monitors industrial equipment would have to be replaced if equipment upgrades/additions result in changes to the software interfaces and functionality. Designing and developing new software agents is costly and time consuming, and is therefore undesirable. VERSAG is able to address this challenge by enabling agents to autonomously search for and acquire software components which implement such new functionality.

Even in non-resource constrained environments, energy conservation is imperative to reduce costs and to be environmentally friendly. Processing capacity is also a limited resource since supply often exceeds the demands of applications. Thus, it is desirable for an agent to have the ability to become lightweight and consume less resources when it has fewer responsibilities (e.g. for a control/monitoring agent when the controlled equipment is idle). The VERSAG framework addresses this challenge in two ways. First, agents have the ability to discard unwanted software components (i.e. capabilities) and become lightweight. Second, they are able to search for and acquire improved implementations of software components as they become available.

7. Conclusion and future work

We have proposed the VErsatile Self-adaptive AGents (VERSAG) framework as a suitable software infrastructure to build ubiquitous computing applications. VERSAG agents, through their adaptive nature, are able to determine which tasks to address, find suitable capabilities with which to fulfil these tasks, and migrate if required. Furthermore, they are able to operate in resource constrained environments and display the scalability inherent to the agent paradigm. Thus, the VERSAG framework exhibits the desirable features of a software infrastructure for ubiquitous computing as identified by Banavar and Bernstein (2002, 2004).

In this article we presented an in depth description of the VERSAG framework and agent architecture. A key contribution of VERSAG is the ability to share functional components (i.e. capabilities) with peer agents, which makes them highly versatile and suitable for ubiquitous computing environments. We also presented design and implementation details of our VERSAG prototype. We note that the formal underpinnings of VERSAG have been previously described in (Gunasekera et al., 2010).

To illustrate the functional benefits VERSAG brings to ubiquitous computing environments, we described a case study scenario implementation where VERSAG agents are used as personal assistant agents. Further empirical evaluations demonstrated that efficient agent migration can be achieved with the aid of peer capability sharing between agents.

In building ubiquitous computing applications with VERSAG, there are challenges related to adaptation decision making, capability management and itinerary generation that are not addressed by the core framework. These need to be addressed at the level of capabilities, and one such example is the multi-criteria decision making cost model we proposed in (Gunasekera et al., 2010). One of our future directions for investigation along these lines is generating detailed agent itineraries from high-level declarative user requests.

Other future directions in which this research could be extended are as follows. At present our prototype implementation is limited to a single agent platform (i.e. JADE). Since the framework has

been developed with the intention of supporting multiple agent platforms and cross-platform agent migration, a natural future progression is to extend support to new platforms and environments. Along the same line of development, we envision that creating a stand-alone version of the framework such that VERSAG agents can execute in environments without any agent infrastructure to be highly valuable. Since there is limited mobile agent toolkit support for hand-held mobile devices like smart phones, personal digital assistants (PDA) and tablets, this will be a first step in expanding VERSAG's reach to such devices. Upgrading the capability model to support the latest OSGi release is another implementation focused future direction of our research.

Acknowledgements

The first author acknowledges support received from the Monash University Postgraduate Publications Award.

References

Alberola, J.M., Such, J.M., Garcia-Fornes, A., Espinosa, A., Botti, V., 2010. A performance evaluation of three multiagent platforms. Artificial Intelligence Review 34, 145–176.

Amara-Hachmi, N., Fallah-Seghrouchni, A.E., 2005. Towards a generic architecture for self-adaptive mobile agents. In: European Workshop on Adaptive Agents and Multi-Agent Systems, Paris, France.

Banavar, G., Bernstein, A., 2002. Software infrastructure and design challenges for ubiquitous computing applications. Communications of the ACM 45, 92–96.

Banavar, G., Bernstein, A., 2004. Challenges in design and software infrastructure for ubiquitous computing applications. Advances in Computers 62, 179–202.

Barton, J.J., Cerqueira, R., Fontoura, M., 2004. Ubiquitous computing. Journal of Systems and Software 69, 207.

Bellifemine, F., Caire, G., Poggi, A., Rimassa, G., 2003. JADE – a white paper. TILAB Journal "EXP – in search of innovation" Special issue on JADE 3, 6–19.

Bellifemine, F., Caire, G., Trucco, T., Rimassa, G., Mungenast, R., 2007a. JADE Administrator's Guide, p. 37.

Bellifemine, F.L., Caire, G., Greenwood, D., 2007b. Developing Multi-agent Systems with JADE. John Wiley and Sons, Chichester, UK.

Brandt, R., 2001. Dynamic Adaptation of Mobile Code in Heterogeneous Environments Institute of Informatics. Technical University of Munich, Munich, p. 79.

Braun, P., Rossak, W., 2004. Mobile Agents Basic Concepts, Mobility Models and the Tracy Toolkit. Morgan Kaufmann Publishers.

Braun, P., Trinh, D., Kowalczyk, R., 2005. Integrating a new mobility service into the Jade agent Toolkit. In: Karmouch, A., Pierre, S. (Eds.), Mobility Aware Technologies and Applications: Second International Workshop, MATA 2005, Montreal, Canada, October 17-19, 2005. Proceedings. Springer-Verlag, Heidelberg, pp. 354–363.

Cardoso, R.S., Kon, F., 2002. Mobile agents: a key for effective pervasive computing. In: ACM OOPSLA 2002 Workshop on Pervasive Computing, Seattle, Washington, USA.

Carzaniga, A., Picco, G.P., Vigna, G., 2007. Is code still moving around? Looking back at a decade of code mobility. In: 29th International Conference on Software Engineering: (ICSE'07 Companion), IEEE Computer Society, Minneapolis, MN, USA, pp. 9–18.

Chen, Q., Chundi, P., Dayal, U., Hsu, M., 1999. Dynamic agents. International Journal of Cooperative Information Systems 8, 195–223.

Choi, S., Choo, H., Baik, M., Kim, H., Byun, E., 2009. ODDUGI: ubiquitous mobile agent system. In: Gervasi, O., Taniar, D., Murgante, B., Laganà, A., Mun, Y., Gavrilova, M.L. (Eds.), Computational Science and Its Applications – ICCSA 2009 International Conference, Seoul, Korea, June 29-July 2, 2009. Proceedings, Part II. Springer-Verlag, Berlin Heidelberg, pp. 393–407.

Chu, H.-h., Song, H., Wong, C., Kurakake, S., Katagiri, M., 2004. Roam, a seamless application framework. Journal of Systems and Software 69, 209–226.

Councill, B., Heineman, G.T., 2001. Definition of a Software Component and Its Elements Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing, Boston, MA, USA, pp. 5–19.

Dixon, K.R., Pham, T.Q., Khosla, P.K., 2000. Port-based adaptable agent architecture. In: Robertson, P., Shrobe, H., Laddaga, R. (Eds.), Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000. Revised Papers. Springer-Verlag, pp. 181–198.

Emmerich, W., Kaveh, N., 2002. Component technologies: Java Beans, COM, CORBA, RMI, EJB and the CORBA Component Model. In: 24th International Conference on Software Engineering (ICSE 2002). ACM Press, Orlando, Florida, USA, pp. 691–692.

Erfurth, C., Kern, S., Rossak, W., Braun, P., Leßmann, A., 2008. MobiSoft: networked personal assistants for mobile users in everyday life. In: Klusch, M., Pechoucek, M., Polleres, A. (Eds.), Cooperative Information Agents XII 12th International Workshop, CIA 2008, Prague, Czech Republic, September 10-12, 2008. Proceedings. Springer-Verlag, pp. 147–161.

Feng, Y., Cao, J., Lau, I.C.H., Liu, X., 2008. A self-configuring personal agent platform for pervasive computing. In: IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC'08), IEEE Computer Society, Shanghai, China, pp. 438–444.

FIPA, 2011. The Foundation for Intelligent Physical Agents. Accessed February 2011. http://www.fipa.org

Gorton, I., 2006. Essential Software Architecture. Springer-Verlag, Berlin Heidelberg.

Gunasekera, K., Krishnaswamy, S., Loke, S.W., Zaslavsky, A., 2009a. Runtime efficiency of adaptive mobile software agents in pervasive computing environments. In: ACM International Conference on Pervasive Services (ICPS'09). ACM Press, London, UK, pp. 123–132.

Gunasekera, K., Loke, S.W., Zaslavsky, A., Krishnaswamy, S., 2009b. Runtime adaptation of multiagent systems for ubiquitous environments. In: 2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2009), IEE Computer Society, Milan, Italy, pp. 486–490.

Gunasekera, K., Zaslavsky, A., Krishnaswamy, S., Loke, S.W., 2009c. Component based approach for composing adaptive mobile agents. In: Håkansson, A., Nguyen, N.T., Hartung, R.L., Howlett, R.J., Jain, L.C. (Eds.), Agent and Multi-Agent Systems: Technologies and Applications 3rd KES International Symposium, KES-AMSTA 2009, Uppsala, Sweden, June 3-5, 2009. Proceedings. Springer-Verlag, Heidelberg, pp. 90–99.

Gunasekera, K., Krishnaswamy, S., Loke, S.W., Zaslavsky, A., 2010. Adaptation support for agent based pervasive systems. In: 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2010), Sydney, Australia.

Gunasekera, K., Loke, S., Zaslavsky, A., Krishnaswamy, S., 2011. Improving efficiency of service oriented context-driven software agents. Cybernetics and Systems 42, 324–340.

Kaelbling, L.P., Littman, M.L., Moore, A.W., 1996. Reinforcement learning: a survey. Journal of Artificial Intelligence Research 4, 237–285.

Kephart, J., Chess, D., 2003. The vision of autonomic computing. Computer 36, 41–50.

Lange, D.B., Oshima, M., 1999. Seven good reasons for mobile agents. Communications of the ACM 42, 88–89.

Marples, D., Kriens, P., 2001. The open services gateway initiative: an introductory overview. IEEE Communications Magazine 39, 110–114.

Milojicic, D.S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., Xu, Z., 2002. Peer-to-Peer Computing. HP Laboratories, Palo Alto, USA, p. 51.

Muldoon, C., O'Hare, G.M.P., Bradley, J.F., 2007. Towards reflective mobile agents for resource-constrained mobile devices. In: 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07). ACM Press, Honolulu, Hawai, pp. 932–934.

Niemelä, E., Latvakoski, J., 2004. Survey of requirements and solutions for ubiquitous software. In: 3rd International Conference on Mobile and Ubiquitous Multimedia (MUM2004). ACM Press, College Park, Maryland, pp. 71–78.

OSGi, 2003. OSGi Service Platform, Release 3. IOS Press, Inc., Amsterdam, The Netherlands.

OSGi, 2007 About the OSGi Service Platform: Technical Whitepaper. OSGi Alliance, pp. 1–19.

Parakh, G., Paprzycki, M., Nistor, C.E., 2002. Dynamically loaded reasoning models in negotiating agents. In: 3rd European Conference on E-Commerce, E-Activities, E-Working, E-Business, E-Learning, E-Health, On-line Services, Virtual Institutes, and their Influences on the Economic and Social Environment (E-Comm-Line), Bucharest, Romania, pp. 199–203.

Picco, G.P., 1998. μCode: a lightweight and flexible mobile code toolkit. In: Rothermel, K., Hohl, F. (Eds.), Mobile Agents: Second International Workshop, MA'98 Stuttgart, Germany, September 9-11, 1998 Proceedings. Springer-Verlag, Berlin Heidelberg, pp. 160–171.

Price, R., Krishnaswamy, S., Arora, N., 2007. Current research in conceptual modelling of agent mobility: an ontology-based evaluation. International Journal of Metadata, Semantics and Ontologies 2, 79–93.

Rellermeyer, J.S., Alonso, G., 2007. Concierge: a service platform for resource-constrained devices. In: 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems. ACM Press, Lisbon, Portugal, pp. 245–258.

Richardson, L., Ruby, S., 2007. RESTful Web Services, First ed. O'Reilly Media, Sebastopol, CA, USA.

Saha, D., Mukherjee, A., 2003. Pervasive computing: a paradigm for the 21st century. Computer 36, 25–31.

Salehie, M., Tahvildari, L., 2009. Self-adaptive software: landscape and research challenges. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 4 14, 01–42.

Satyanarayanan, M., 2001. Pervasive computing: vision and challenges. IEEE Personal Communications 8, 10–17.

Spyrou, C., Samaras, G., Pitoura, E., Evripidou, P., 2004. Mobile agents for wireless computing: the convergence of wireless computational models with mobile-agent technologies. Mobile Networks & Applications 9, 517–528.

Steinmetz, R., Wehrle, K., 2005. What is this "Peer-to-Peer" about? In: Steinmetz, R., Wehrle, K. (Eds.), Peer-to-Peer Systems and Applications. Springer-Verlag, Berlin Heidelberg, pp. 9–16.

Tu, M.T., Griffel, F., Merz, M., Lamersdorf, W., 1998. A plug-in architecture providing dynamic negotiation capabilities for mobile agents. In: Rothermel, K., Hohl, F. (Eds.), Mobile Agents: Second International Workshop, MA'98 Stuttgart, Germany, September 9-11, 1998 Proceedings. Springer-Verlag, Heidelberg, pp. 222–236.

Voyager, 2007. Voyager Edge. Accessed November 2007. http://www.recursionsw.com/Products/voyager.html

Zhang, D., Wan, J., Liu, Q., Guan, X., Liang, X., 2012. A taxonomy of agent technologies for Ubiquitous computing environments. KSII Transactions on Internet and Information Systems 6, 547–565.

Kutila Gunasekera received his PhD in Information Technology from Monash University, Australia in 2012. He also holds an MSc in Computer Science (2006) and a BSc in Engineering (2002) from University of Moratuwa, Sri Lanka. Kutila's research interests are in the areas of mobile and pervasive computing, mobile agents and the semantic web. He currently works as a software engineer with the e-research group in University of Queensland.

Professor Arkady Zaslavsky is Science Leader of Semantic Data Management at the CSIRO ICT Centre. Previously, he was Chaired Professor in Pervasive and Mobile Computing at Luleå University of Technology, Sweden and a full-time academic staff member at Monash University, Australia from 1992 to 2008. Arkady received MSc in Applied Mathematics from Tbilisi State University (Georgia, USSR) in 1976 and PhD in Computer Science from the Moscow Institute for Control Sciences (IPU-IAT), USSR Academy of Sciences in 1987. Arkady has published more than 300 research publications throughout his professional career and supervised to completion more than 30 PhD students.

Shonali Krishnaswamy is Deputy Head of the Data Mining Department at the Institute for Infocomm Research (I2R), Singapore. She is also an Associate Professor at Monash University, Australia. Her research interests are in the areas of Mobile, and Distributed Data Mining, and Real-time Data Stream Mining. She is increasingly interested in Energy Analytics, and Mobile User Analytics. Shonali has published around 150 research papers and has been the recipient of many awards including: Monash University Vice-Chancellors Award for Excellence in Research by an Early Career Researcher, IBM Innovation Award, and an Australian Post Doctoral Fellowship from the Australian Research Council.

Seng Wai Loke is a Reader and Associate Professor at the Department of Computer Science and Computer Engineering in La Trobe University. He leads the Pervasive Computing Group at La Trobe, and has authored 'Context-Aware Pervasive Systems: Architectures for a New Breed of Applications' published by Auerbach (CRC Press), 2006. He has (co-)authored more than 220 research publications including 40 journal papers, 10 book chapters, and over 150 conference/workshop papers, with numerous work on context-aware computing, and mobile and pervasive computing. He has been on the program committee of numerous conferences/workshops in the area, including Pervasive'08 and Percom'10 (and 2011).