Copyright Notice

Compositionally Adaptive Mobile Software Agents for Pervasive Environments

Kutila Perera Rajakaruna Gunasekera

BSc (Eng), MSc (Computer Science)

Thesis

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

Caulfield School of Information Technology

Monash University

September 2011

Declaration

I hereby declare that this thesis contains no material which has been accepted for the award of any other degree or diploma in any university or equivalent institution, and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Kutila Perera Rajakaruna Gunasekera

16 December 2011

Acknowledgements

The guidance, encouragement and assistance of many people helped me through the ups and downs of doing a PhD. While it is impossible to name all of them, I would like to take the opportunity to thank and acknowledge the following people.

First, I would like to thank my supervisors Prof. Arkady Zaslavsky, A/Prof. Shonali Krishnaswamy and A/Prof. Seng Loke. I am grateful to Arkady for accepting me as a PhD candidate, for his insights, guidance and belief in my ability even when mine wavered. I am deeply grateful to Shonali for her invaluable ideas, attention to detail and detailed feedback which helped shape this thesis. I convey my gratitude to Seng, for his many ideas on how to improve my research and prompt responses and feedback on my numerous queries.

I am grateful to Prem Jayaraman and Sudanthi Wijewickrema for their help in proof-reading the thesis. I would like to thank Allison Mitchell, Julie Austin, Katherine Knight, Michelle Ketchen, Rob Gray, Cornelia Lioulios, Denyse Cove, Duke Fonias, See Ngieng, Diana Sussman, and Akamon Kunkongkapun for always being helpful and providing me with administrative and technical support during my research.

I thank my colleagues and fellow PhD students Prem Jayaraman, Waskitho Wibisono, Yong Bin Kang, Ruwini Kodikara, Brett Gillick, Sunam Pradhan, Pari Delir Haghighi, Karan Mitra, Saguna, Abdullah Almuhaideb, Peter Serwylo, Luke Steller, Colin Enticott and other DSSE members who made the candidature period more enjoyable and helped in numerous ways.

I would also like to thank Chris Ling, Maria Indrawan, Jeff Tan, Philip Chan, Glenn Jayaputera and Damminda Alahakoon for their assistance in numerous capacities.

I owe a great deal to Nishantha and Niranjee Rajapaksha for helping me settle in Australia and practically providing a home away from home. The friendship of Chamindra Lakmali Vithana (Laki), Sudanthi Wijewickrema, Ishan Hettiarachchi and Dhananjay Thiruvady also helped make life enjoyable during the lonely journey

towards a PhD. I also thank the Buddhist Society of Victoria for providing a safe haven that I could escape to whenever I needed a break.

I must thank Ms Vishaka Nananayakkara, Prof. Gihan Dias, Prof. Dileeka Dias, Assistant Prof. G. C. De Silva and the staff at the Department of Computer Science and Engineering at University of Moratuwa for encouraging me to pursue a PhD and prior to that taking me to a level where I could do one! Himath Dissanayake, long time friend, mentor, team captain and manager, deserves special mention for it is from him that I picked up many skills that have helped me along the journey.

I owe a great deal to my best friends Punya Keerthi and Radesh Batuwita for being such wonderful friends, believing in my ability when I lost confidence, and for filling in as sons to my parents when I was not available during their times of need.

I am eternally grateful to my brothers Duminda and Pankaja, especially loku aiya, for looking after our parents and giving me the freedom to be away from home. Finally, I would like to thank my parents, for their unconditional support, love and belief. It is to them that I owe everything I am.

To the memory of my father, who left us in 2010

Abstract

The vision of pervasive computing foresees environments filled with multiple computing devices unobtrusively aiding humans in their daily activities. With recent advances in hardware and communication technologies, computing capacity is becoming increasingly pervasive and invisible in our environment. The continuing challenge is to achieve seamless and graceful integration of these pervasive technologies with our daily activities, which is now the increasing focus of research within this area.

Mobile agents are lightweight software entities that move from one node to another in a network autonomously and continue execution of their own accord. Mobile software agents are seen as a suitable enabling technology for pervasive computing because they bring flexibility, scalability and the ability to simplify tasks by movement and delegation for a range of pervasive applications. However, mobile agent based applications have limited adaptability, and this is an inhibiting factor in pervasive environments. The inherent nature of pervasive environments is that they are complex, heterogeneous and highly dynamic. This makes it extremely hard to design and program mobile agent systems that cater for all possible situations that could be encountered in such environments.

This thesis investigates, proposes and develops a novel platform for building adaptive software agents which combines mobility with the ability to dynamically change the internal structure and capabilities, as a strategy to harness the potential of mobile agents for pervasive environments. We term this *compositional adaptation of mobile agents* and focus on enhancing the adaptation potential of mobile agents for the purpose of building applications and services for pervasive environments.

The first contribution of this thesis is our proposed VErsatile Self-adaptive AGents (VERSAG) framework. The framework is centred on a component-based agent model where an agent is a lightweight mobile entity whose application-specific functionality is provided in the form of reusable software components termed *capabilities*. Furthermore, an agent can increase its autonomy at runtime through

acquisition (and potential discarding) of capabilities which improve its core functions such as reasoning ability and context-awareness. Our second contribution is the peer capability sharing feature of VERSAG agents. That is, when an agent does not contain required capabilities, it is able to search for, select and acquire them from other agents who are willing to share their capabilities. Our third contribution achieves cost-efficient adaptation of agents using a proposed cost model that allows agents to make adaptation decisions by taking into consideration multiple cost criteria. These cost criteria can be either user/organization specified or based on contextual constraints. We use time, network usage, computational resource consumption and accuracy of results, as representative cost criteria.

In this thesis we propose, implement and evaluate the conceptual framework of VERSAG and its theoretical foundations. We have developed a prototype implementation of VERSAG to demonstrate its feasibility and to conduct experimental evaluation of the concepts that underpin the proposed framework. The feasibility and functional benefits achieved from this approach are demonstrated using an application case study. Our experimental evaluation also demonstrates that VERSAG agents bring about performance benefits over conventional mobile agents through the use of their capability sharing feature. Through extensive experimental evaluation we also establish the scalability of component sharing mobile agents and the effective performance of the decision making cost model. Thus, this thesis takes a step forward in realising the potential of mobile software agents enhanced with compositional adaptation for pervasive computing. The research work described in this thesis has resulted in one international journal article and seven peer-reviewed international conference papers.

Thesis Work Outcomes

This thesis has resulted in eight peer-reviewed publications. These publications include one journal article and seven international conference papers.

Journal

1. Gunasekera, K., Loke, S., Zaslavsky, A. and Krishnaswamy, S., (2011), *"Improving Efficiency of Service Oriented Context-Driven Software Agents"*, Journal of Cybernetics and Systems, Vol. 42, No. 5, pp. 324-340.

International Conferences

2. Gunasekera, K., Krishnaswamy, S., Loke, S. W. and Zaslavsky, A., (2010), *"Adaptation Support for Agent Based Pervasive Systems"*, Proceedings of the 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2010), Sydeny, Australia, 6-9 December. (acceptance 23% for regular papers)
3. Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2010), *"Service Oriented Context-Aware Software Agents for Greater Efficiency"*, 4th International KES Symposium on Agents and Multi-agent Systems – Technologies and Applications (KES-AMSTA 2010), Gdynia, Poland, Agent and Multi-Agent Systems: Technologies and Applications, Lecture Notes in Computer Science, 6070, Springer-Verlag, pp. 62-71.
4. Gunasekera, K., Loke, S. W., Zaslavsky, A. and Krishnaswamy, S., (2009), *"Runtime Adaptation of Multiagent Systems for Ubiquitous Environments"*, Proceedings of the 2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2009), Milan, Italy, 15-18 September, IEEE Computer Society, pp. 486-490. (acceptance 18% for regular papers and 24% for short papers)
5. Gunasekera, K., Krishnaswamy, S., Loke, S. W. and Zaslavsky, A., (2009), *"Runtime Efficiency of Adaptive Mobile Software Agents in Pervasive Computing Environments"*, Proceedings of the ACM International Conference

on Pervasive Services (ICPS'09), London, UK, 13-16 July, ACM Press, pp. 123-132. (Best paper award)

6. Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2009), *"Component Based Approach for Composing Adaptive Mobile Agents"*, 3rd International KES Symposium on Agents and Multi-agent Systems – Technologies and Applications (KES-AMSTA 2009), Uppsala, Sweden, Agent and Multi-Agent Systems: Technologies and Applications, Lecture Notes in Computer Science, 5559, Springer-Verlag, pp. 90-99.
7. Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2008), *"VERSAG: Context-Aware Adaptive Mobile Agents for the Semantic Web"*, Proceedings of the 3rd IEEE International Workshop on Engineering Semantic Agent Systems (ESAS 2008), at COMPSAC 2008, Turku, Finland, 28 July-1 August, IEEE Computer Society, pp. 521-522.
8. Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2008), *"Context Driven Compositional Adaptation of Mobile Agents"*, Proceedings of the International Workshop on Data Management in Context-Aware Computing (DMCAC 2008) Beijing, China, 27-30 April, IEEE Computer Society, pp. 201-208.

Poster Presentations

- Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2009), *"VERSAG: Versatile Self-adaptive Agents for Pervasive Computing Environments"*, HDR Poster Exhibition, Monash University.
- Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2010), "*VERSAG: Versatile Self-adaptive Agents for Pervasive Environments*", DSSE Industry Day, Monash University.

Contents

List of Figures

List of Tables

C H A P T E R O N E

1 Introduction

1.1 Preamble

The vision of pervasive computing foresees environments filled with multiple computing devices unobtrusively aiding humans in their daily activities. Two decades after Mark Weiser's seminal paper [Wei91], computing devices ranging from tiny sensors to mobile phones, laptop computers, and in-vehicle computers pervade our environment. Wireless communication technologies are commonplace and devices such as sensors and mobile phones are able to communicate with each other using short-range communication technologies such as Infrared (IR), Bluetooth and ZigBee (IEEE 802.15.4) when fixed networking infrastructure is unavailable. Thus, computing capacity has become quite pervasive and at least partially "invisible" in our environment. The continuing challenge is to achieve seamless and graceful integration of these pervasive technologies with our daily activities, which is now the increasing focus of research within this area.

Achieving the goals of pervasive computing requires addressing a multitude of research challenges, some of which have common themes with related disciplines such as distributed and mobile computing. Satyanarayan [Sat01] identifies four major areas of interest in pervasive computing research:

- *Effective use of smart spaces* refers to the ability of the virtual and physical worlds to sense and control each other. For example, the ambience of a hotel room may be automatically adjusted depending on the occupant's preferences

available from his or her virtual profile. Similarly, his or her virtual presence will adjust its behaviour to suit the surroundings, such as encrypting communications since the hotel network is not a trusted domain.

- *Invisibility* refers to the ability to minimize user distraction by being proactive and anticipating user intent.
- *Localized scalability* discusses the need for scalability in pervasive computing devices and the relevance of physical proximity in interactions between devices.
- *Handling uneven conditioning* is important as different environments will have varying degrees of infrastructure support for pervasive computing. Personal computing devices should be able to adapt accordingly and hide these differences from the user so as to maintain *invisibility*.

Satyanarayan [Sat01] further identifies that the hardware, communication and component technologies required by pervasive computing are already available and that the major research problems are to be found in weaving these together to build seamless pervasive applications and services. Ten years on, advances in pervasive computing have been led largely by developments in hardware and communication technologies, and the ability to build software systems which can seamlessly make use of these devices and communication technologies is an active and important research challenge. Heterogeneity of devices and communication technologies, limitations in connectivity, computing capacity and power, and rapidly varying environments are some of the challenges faced when developing applications and services leveraging such infrastructure [NiL04]. Approaches such as *autonomic computing* [KeC03] and *self-adaptive software* [Lad97] have been proposed for developing pervasive applications and services, and *mobile and intelligent software agents* are often featured within these approaches [CaK02, Sat01].

Mobile agents are considered a useful approach for pervasive computing because they bring flexibility, ability to simplify tasks by delegation and scalability to pervasive applications [BrR04, OgO05, Zas04]. Coupled with the intelligent agent paradigm, mobile agents can support pervasive devices with intelligence as well as dynamic software components that can work both independently and as a part of a

larger distributed system. While mobile agent based applications are adaptable by nature, it is extremely difficult, if not impossible, to design and program an agent system to cater for multiple situations it may encounter in a complex and dynamic environment [AKK03, MaM06]. Thus, a key aspect of ongoing research is aimed towards solving issues related to modelling and building of adaptive agent systems [MaM06, Pit90].

This thesis focuses on harnessing and enhancing the potential of mobile agents for the purpose of building applications and services for pervasive environments. Towards this end, we propose and develop VERSAG, a novel platform for building adaptive software agents which combines mobility with compositional adaptation, thus enhancing and revealing the potential for adaptability that mobile agents possess.

The rest of this chapter is organized as follows. Section 1.2 provides an overview of mobile software agents and is followed in section 1.3 by a discussion of their suitability for pervasive computing. Then, section 1.4 provides a brief overview of agent adaptation. The motivation and objectives of the research are described in sections 1.5 and 1.6. The research scope and contributions are outlined in section 1.7 and section 1.8 concludes the chapter with an outline of the thesis organization.

1.2 Software Agents and Mobile Agents

While there is no universally accepted definition of a software agent, the definition by Wooldridge [page 15 of Woo02] provided below captures the essence of agency:

> "*An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives*"

Furthermore, the following four characteristics are accepted as requirements for a software entity to be called an agent [BrR04, Woo00]:

- *Autonomy* is the ability to act without direct human intervention. For example, a user would only specify a high-level goal and the agent should be

able to build a detailed plan for achieving this goal and to execute it, while making intermediate decisions by itself.

- *Social behaviour* is the ability of an agent to communicate with other agents or humans to exchange information and negotiate in order to achieve its goals.
- *Reactivity* is the ability to sense events in the environment and react to those that are relevant to it.
- *Proactivity* is the ability to actively plan and take action in order to reach its goal.

A *mobile agent* is a form of mobile code where a software entity moves from one node to another in a network and continues execution on its own volition [BrR04, Pic01]. Taken from a purely intelligent agent perspective, the ability to migrate an agent from one host to another host is an orthogonal feature of a software agent [page xv of Woo02]. However, from a distributed computing perspective, the ability of self-directed migration is a central feature for building and deploying systems and applications. Figure 1-1 illustrates an agent migrating to a different host in order to consume a service available on the destination host. Mobile agents are also able to migrate multiple times, or demonstrate *multi-hop* ability, which together with agent autonomy sets them apart from other mobile code mechanisms such as remote evaluation and code on demand [Pic01]. A mobile agent may have an associated *itinerary* which specifies a list of hosts for it to travel to and activities to execute at each location [PKA07].

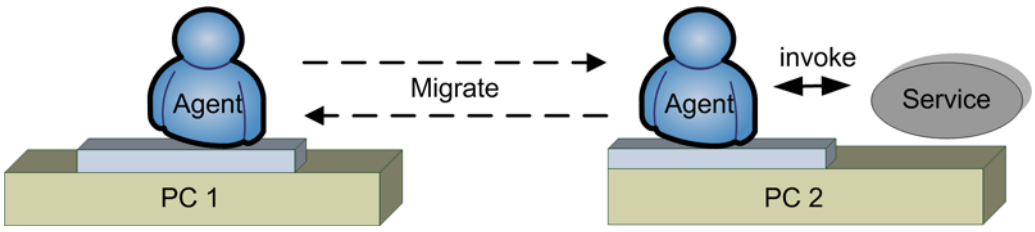


Figure 1-1: Mobile agent migrates to a different host to access a service

As a distributed computing technique mobile agents are complementary to existing techniques such as the client-server paradigm, and as such, can be used together with

traditional approaches to achieve better performance. Four major technical advantages of mobile agents from [BrR04] are listed below:

1. *Delegation of tasks.* Agents are autonomous representatives of the user and therefore can handle tasks without always requiring user intervention. This reduces the amount of user attention needed.
2. *Asynchronous processing.* They are suitable for mobile/disconnected operations as connectivity is only needed to launch an agent from the user's device (i.e. connectivity does not need to be continuous).
3. *Adaptable service interfaces.* Since most distributed services provide generic functional interfaces, mobile agents can be used to aggregate these into more complex services optimized for end users' needs. An agent is thus able to simulate a constant and specialized interface to the user while using different primitive back-end services.
4. *Code shipping Vs data shipping.* This is one of the most visible advantages of mobile agents. A mobile agent moves the code to where the data is, instead of the traditional mechanism of data being moved to the location of the processing code. This approach allows reduced bandwidth consumption and network load when the data is larger than the code (e.g. [Pic01]).

Mobile agents have traditionally been considered beneficial for application areas such as network management, electronic commerce, distributed data mining and information retrieval [BrR04, LaO99, Mil99, Pic01], and recently they are also being considered as an attractive option for pervasive computing [BrR04, CaK02, LaO99, Pic01, Sat01]. In the next section we describe several key reasons why mobile agents are seen as suitable for pervasive computing.

1.3 Mobile Agents for Pervasive Computing

Mobile agents are a suitable approach for pervasive computing, and recent research efforts such as the ODDUGI mobile agent platform [CCB09], the Irish Agent Factory [MOB07] and MobiSoft [EKR08] are representative of efforts to harness this suitability. This sub-section outlines how mobile agents can assist with several key

features required of pervasive applications, namely heterogeneity, mobility, context-awareness and adaptability:

- *Heterogeneity.* The pervasive computing world can be divided into discrete environments, often based on physical world boundaries such as a home, a conference room or a vehicle, and entities can dynamically enter and leave these environments [KiF02]. For example, when a user enters a "smart home" [EdG01], his or her PDA should join the home network and be able to interact with other entities in the home. Pervasive applications and services have to be accessed over diverse communication mechanisms, depending on availability of network infrastructure and radio capabilities of devices (e.g. a smart phone may be equipped with wireless LAN, GSM and Bluetooth radio whereas a sensor device is likely to only support a single wireless radio technology). Also, these devices are mostly everyday objects that have been enhanced with computing capacity in addition to their main functionality (e.g. MediaCup [BGS01]). Often, they are purpose built, battery powered, mobile, have lower processing capacity than desktop computers, have limited input/output mechanisms, and contain diverse hardware and software [SaM03, Sat01, Wei91]. Complex software systems are needed to seamlessly integrate users/entities with these heterogeneous systems and to withstand the dynamic changes observed in pervasive environments. Software agents present a suitable software engineering paradigm in such situations due to their ability to reduce complexity by delegation [Zas04]. For example, managing different types of devices could be delegated to different agents, where each agent handles the peculiarities of that device. Agents can also reduce information overload on human users with their intelligent and autonomous information processing behaviour.
- *Mobility.* Mobility is an integral component of pervasive computing [Sat01] since users/entities move in and out of smart environments carrying typically mobile computating devices. For example, a pervasive application may need to migrate when its hosting device and user move away from each other, when the host device is low on resources, or when network connectivity deterioration adversely affects application functioning [KiL07, PrB08a,

RCC04, YCW06]. Mobile software agents are ideally positioned to fulfil this need as pervasive applications can be implemented using mobile agents, allowing them to autonomously migrate when needed.

- *Context-awareness.* Context-awareness is another important feature for pervasive computing applications [Sat01, Zas04]. Context can be defined as "the set of environmental states and settings that either determines an application's behaviour or in which an application event occurs and is interesting to the user" [ChK00], and includes *computing* (e.g. network parameters, computing resources), *user* (e.g. user preferences, location, activity) and *physical* (e.g. temperature, noise, lighting level) contexts [SAW94]. Being context-aware allows a pervasive application to appropriately react to changes in its environment and to be minimally intrusive to users. A simple example is a smart phone application that senses noise level in its vicinity and automatically adjusts the phone volume. Mobile software agents have been identified in research as a suitable approach for enabling context-awareness in pervasive systems [CFJ03, RaC03, Sat10, Zas04]. Zaslavsky [Zas04] further argues that mobile agents can also be used as context explorers to gain better understanding of pervasive environments.
- *Adaptability.* A pervasive application needs to adapt in response to events in its environment, such as variations in resource availability (e.g. battery level, network bandwidth), changes in user preference, and appearance and disappearance of devices [CaK02, KiF02, Sat01]. Once again, mobile software agents can fulfil these needs through migration and adapting individually or as a multi-agent system. For example, a pervasive application could adapt to reduced network connectivity by introducing a heavier client so that the lack of connectivity is not noticeable to the user (i.e. replace a thin mobile agent client with a more intelligent client agent that is able to do more tasks by itself without needing access to remote service providers).

The above discussion shows that mobile software agents have strong capabilities to assist with heterogeneity, mobility, context-awareness and adaptability which are important features of pervasive software [NiL04, YCW06]. The next section

describes the concept of agent adaptation and its particular significance in pervasive computing.

1.4 Agent Adaptation for Pervasive Computing

Agents operating in pervasive environments have to interact with heterogeneous entities and adapt accordingly. However, a software agent is usually designed with a fixed set of functionality. Thus, an individual agent is limited in what tasks it can achieve. In complex, dynamic environments, such as those found in pervasive computing, an agent may be faced with situations that require capabilities beyond its design-phase skill set. In such situations, the agent would have to be replaced with another agent that contains the necessary skills. Furthermore, a context-aware mobile agent in a pervasive environment may have to migrate to a different host as a result of resource constraints on its current device (e.g. low battery). It is possible that agents would encounter new platforms and entities that were not known at agent design time, and hence are unsupported. Thus, mobile agent based applications are able to adapt to dynamic situations with relative ease by moving agents, adding new agents or removing agents from the environment. However, it is extremely hard to design and program an agent system to cater for all possible situations it may encounter in a complex and dynamic environment [AKK03, MaM06]. Therefore, agent systems faced with unexpected situations need to either adapt, be replaced or redesigned. As a result, much research effort in recent times has been exerted on solving issues related to the modelling and building of adaptive agent systems [MaM06, Pit90].

Multiple interpretations of what constitutes an *adaptive agent* can be found in the literature, the most relevant of which are given below:

- Pitrat [Pit90] states that an agent needs to be aware of its structure and evolutionary capabilities (meta-knowledge) to be adaptive.
- Maes [Mae94] considers an agent as being adaptive *"if it is able to improve over time, i.e. if the agent becomes better at achieving its goals with experience."*

- Following along the lines of Pitrat and Maes, Guessoum [Gue04] states that adaptable agents are aware of their own capabilities and can change their structure and knowledge at runtime in order to remain useful.
- Splunter, Wijngaards and Brazier [SWB03] interpret adaptation as *"structural changes of an agent, including knowledge and facts available to an agent."*
- According to Amara-Hachmi and Fallah-Seghrouchni [AmF05] agents become self-adaptive when they have the capability of dynamically reconfiguring their internals (i.e. components and parameters) continuously to fit the environment.

In the following instances agents with adaptive features have also been referred to as *dynamic agents*.

- Zhao, Mao and Wang [ZMW06] consider an agent *"capable of adapting to different behaviours at run-time and playing different roles in various organization contexts"* to be a dynamic agent.
- Chen, Chundi et al. [CCD99] define their dynamic agents as those which can change their behaviour while executing.

Three common features that can be seen from the above interpretations are that an adaptive agent:

- is aware of its structure and ability to change at runtime;
- makes use of this self-awareness to change its internals at runtime (i.e. agent structure, knowledge/data contents); and
- is able to improve its performance over time (or maintain its usefulness in a changing environment) as a result of adaptation.

A multi-agent system (MAS), which consists of multiple agents interacting in an environment can also be adaptive. Marín and Mehandjiev [MaM06] define an adaptive multi-agent system (AMAS) as being *"situated in an open environment and capable to self-modify its structure and internal organization by varying its elements' interactions according to environmental changes."* Guessoum [Gue04] states that an adaptive MAS *"is an open system that dynamically and continuously modifies its structure."* A system is considered open if agents can join and leave the system at

runtime. A similar definition is used to interpret an Open MAS by Vercouter [Ver01] who considers a multi-agent system that supports the addition and removal of agents and the internal capability modification of agents to be an Open MAS. In summary, an adaptive MAS should be open (i.e. agents can be added and removed at runtime) and capable of changing its behaviour at runtime to suit the environment.

The adaptation seen in the whole MAS is usually a result of changes in organizing the agents and their interactions. Individual agents in adaptive MAS may or may not be adaptive. For example, in the AMAS theory [CGG03] individual agents are not adaptive. A multi-agent system can thus be adaptive without its constituent agents falling into the category of adaptive agents.

Agent adaptation and learning are two terms which go hand in hand [DPK00, Pit90, Wei96, Woo02]. Learning is about optimizing agent behaviour by selecting from a set of available actions to suit a set of environmental situations. An agent can learn individually or as a group with the aid of other agents. Maes [Mae94] discusses agent adaptation to be a result of learning while Weiß [Wei96] considers adaptation to be enclosed within the learning process. Agent learning can therefore be considered an approach to adaptation in agent applications.

Thus, agent adaptation can occur either at multi-agent level or at single agent level. At the MAS level, interactions between agents, multi-agent learning and adding/removing agents from a system lead to adaptation. At the single agent level, learning by a single agent and dynamically composing internal features of an agent can lead to adaptation. These approaches (shown in Figure 1-2) are not mutually exclusive and are often used concurrently to achieve better outcomes. It should be noted that while agent mobility has not been included in the classification in Figure 1-2, it is a key form of adaptation and is a central theme of the research discussed in this thesis.

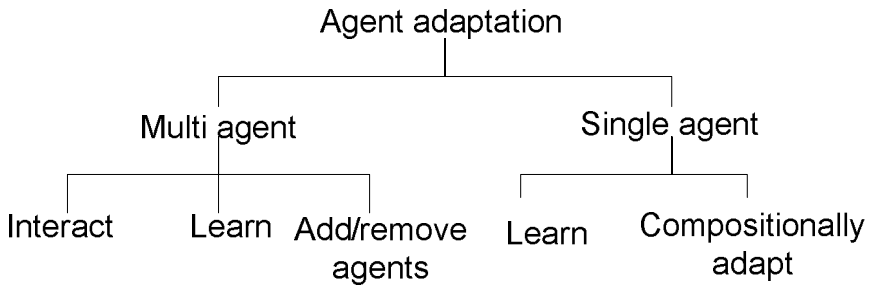


Figure 1-2: Taxonomy of agent adaptation

An important point to note here is that while adaptive MAS [AKK03, CGG03, Gue04, MaM06] and agent learning [PaL05, Wei96] have been the focus of a considerable body of research over time, *dynamic compositional adaptation* of agents has received little attention. A software application adapts *compositionally* when its internal structure is changed by acquiring new components [MSK04b]. It is a powerful form of adaptation since it allows agents to radically change their behaviours while maintaining identity. Thus, with dynamic (i.e. runtime) compositional adaptation, agents are no longer limited to a fixed skill set and can gain completely new behaviours to handle situations which were not foreseen at design time.

For compositional adaptation to be useful, an agent needs to be able to describe diverse high-level requests it may receive as combinations of smaller activities that can be fulfilled using software components. An agent then needs to be able to find suitable software components to fulfil these activities. It is also imperative that goal fulfilment through this agent adaptation and subsequent component execution process is achieved in a cost-efficient manner, where cost may be measured using multiple criteria. For example, a request may have to be completed within a given time, or there may be restrictions due to low powered computing devices and poor connectivity, two situations that are common in pervasive computing scenarios. Reusability of software components also significantly increases cost-efficiency of the overall approach.

We have established that software agents are a suitable abstraction for pervasive computing, and that these agents need to be mobile and compositionally adaptive. In this thesis we investigate a novel approach to develop compositionally adaptive

mobile software agents that are suitable for the varying conditions encountered in pervasive environments. The research motivations and objectives are described next.

1.5 Research Motivations

The preceding discussion brought into focus that building pervasive software is a current challenge in pervasive computing, and that the mobile agent paradigm is seen as a suitable solution to this challenge. It was also seen that state-of-the-art mobile agents are not sufficiently adaptive to deal with the varying conditions that may be encountered in pervasive computing scenarios. We describe next, two hypothetical scenarios where mobile agents and their adaptivity are beneficial.

Scenario 1: Ubiquitous Personal Assistant Agent

We consider using mobile agents as ubiquitous and virtual personal assistants of human users (as in [EKR08]). A personal assistant agent encapsulates a profile of the user including details such as his or her identity, preferences, personal calendar, reminders, and is the user's representative in the virtual pervasive world. The agent[1] can "live" on computing devices that its owner uses (e.g. smart phone, laptop computer, desktop PC) and also migrate to other devices when required as shown in Figure 1-3. It is desirable that this agent is long-lived and capable of carrying out different tasks on behalf of its owner.

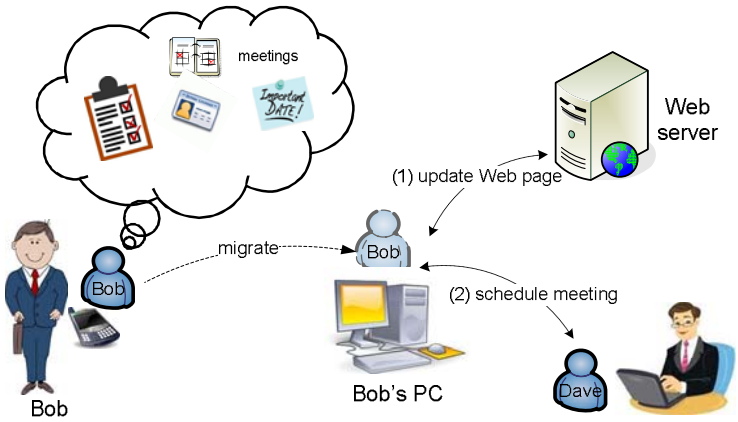


Figure 1-3: Scenario of a ubiquitous personal assistant agent

[1] We use the terms "agent" and "mobile agent" interchangeably except where explicitly mentioned otherwise.

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 desirable in such situations that the agent adapts itself by dynamically loading and shedding functionality based on needs. Then, the agent could be lightweight when it resides on the user's smart phone or similar resource constrained devices. When the user provides it with a task, the agent could migrate to a fixed computer on the wired network, acquire necessary skills and carry out the requested task.

Scenario 2: Mobile Agents at work - Exploring Context

In the aftermath of an earthquake, context exploring mobile agents (as proposed in [Zas04]) are deployed to roam computing devices located within the disaster area to gather information in order to help recovery and rescue missions (as depicted in Figure 1-4). The agents are required to mine interesting data from sensors available on devices they encounter (e.g. accelerometer, light sensor, GPS, proximity sensor), build a view of the environment and send it back to their command centre. With wired infrastructure and power supply likely to be disrupted, agents would mostly have to travel on wireless and ad hoc networks connecting mobile devices, mobile rescue robots [Dav02] and any remaining stationary computing devices. Since most of these devices would be battery powered, it is necessary to be conservative in data transmission, processing as well as the number of agents used. The processing required on each device can vary significantly due to the heterogeneity of device platforms and sensors. Thus, an agent would need to be able to access different types of sensor interfaces as well as adjust the granularity of its mining algorithms according to available resources (i.e. battery power) [SGB10]. However, it is not desirable to have one mobile agent per each type of processing or to have "large" mobile agents with multiple types of processing as both would increase data transmission costs. Thus, it is ideal if agents could adapt themselves, acquiring necessary skills from peer agents and discarding used and unwanted skills to remain lightweight.

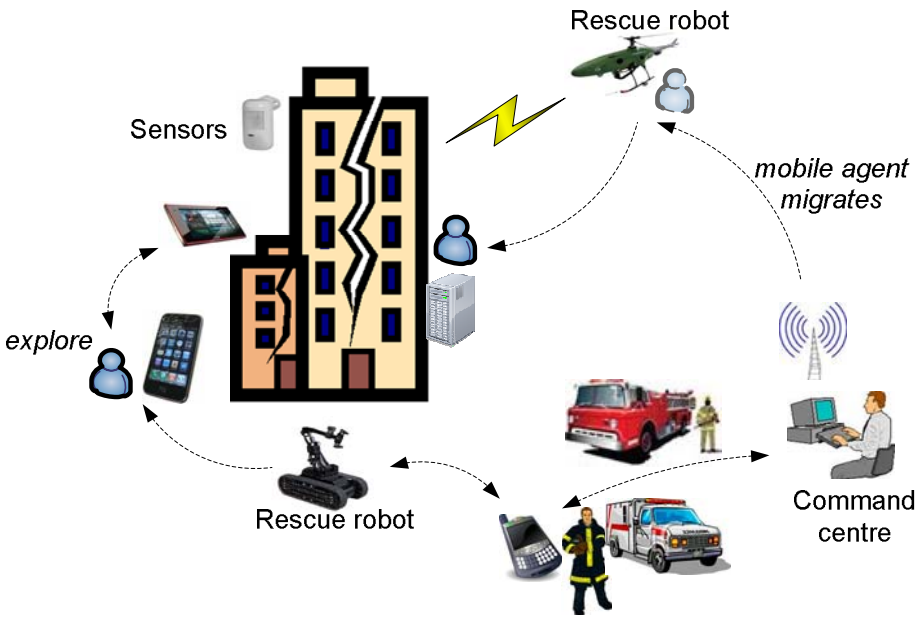


Figure 1-4: Scenario of context gathering mobile agents in a disaster zone

Dynamic compositional adaptation of agents coupled with mobility is clearly a suitable approach in both above scenarios. When building solutions for such scenarios, it is necessary to address several issues related to agent adaptation:

- Agent functionality needs to be represented as software components that can be attached to and detached from agents at runtime. These components should be reusable in order to be of value in the long term.
- 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.
- Consequentially, components should be described in such a way so as to enable searching for them and reasoning about their suitability for a given task, environment and constraints such as trustworthiness and resource needs.
- An agent needs to be able to expand a high-level goal supplied by a user to a detailed set of activities that can be carried out with the aid of a combination of software components. For a mobile agent, this could be in the form of an itinerary specifying a list of locations to visit and components to execute at each location.

- 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. For a compositionally adaptive mobile agent, adaptation activities include migration, acquiring new components and discarding constituent components.

The adaptation process needs to take into account multiple criteria such as contextual conditions, user preferences, application requirements and available alternatives. Therefore, these agents need to possess sophisticated adaptation decision making techniques to perform their tasks in a cost-efficient manner.

1.6 Aims and Objectives

The main aim of this thesis can be stated as follows:

> *To carry out research, development and validation in order to prove that mobile software agents equipped with compositional adaptation capabilities are a suitable approach to build and deploy cost-efficient applications for pervasive computing environments.*

By cost-efficiency we refer to the ability of an agent to select the least cost alternative when presented with multiple alternatives which can achieve a given goal. Overall cost can be measured as an aggregation of multiple criteria such as time, network load and computational resources depending on user preferences, policies and the computing environment.

To achieve these objectives, this thesis proposes, develops and evaluates VERsatile Self-adaptive AGents (VERSAG), an enabling platform for pervasive environments which encompasses cost-efficient and dynamically adaptive mobile software agents. It focuses on the research issues in compositional adaptation of mobile software agents identified earlier.

1.7 Scope and Contributions

This thesis proposes and develops a novel enabling platform to build smart pervasive applications and services through the use of compositionally adaptive mobile

software agents. Our proposed VERSAG (VERsatile Self-adaptive AGents) framework allows agents to execute on heterogeneous environments and to adapt in the face of previously unknown circumstances by migrating, discarding its constituent components or by acquiring new software components from peer agents.

In order to achieve the objectives stated in the previous section, this thesis makes the following contributions:

- This research proposes, designs and implements a component based framework and agent architecture for building compositionally adaptive mobile software agents suited for pervasive environments. The functional and performance benefits of this proposal are validated through a prototype implementation and empirical evaluations.
- The concept of sharing functional components among peer agents is proposed, implemented and validated as a mechanism to increase efficiency and utility of mobile agents in pervasive environments.
- We propose, implement and validate a cost model that allows agents to make cost-efficient adaptation decisions taking into consideration multiple cost criteria, user specified constraints as well as contextual conditions. Time consumption, network usage, computational resource consumption and accuracy of results have been used as representative cost criteria.

The scope of this research is restricted by the following assumptions:

- Security and trust are long-running research challenges in the field of mobile agents. The VERSAG approach shares these challenges as well as new challenges introduced by its use of reusable and shareable software components. Investigating security issues are beyond the scope of this thesis. However, we believe that theory and foundations from the large body of existing research on mobile agent security [Bor02, Jan00, PZI04, Sse10] can be used to secure VERSAG agents.
- This research does not focus on rationality of agents and the multi-agent notions of negotiation and cooperation between agents. From the perspective

of VERSAG, these features can be implemented as reusable components that agents can execute.

- At present, there is limited support for deploying mobile agents on existing devices such as mobile phones and personal digital assistants (PDA), due to a lack of required functionality in their software development environments. These limitations exist more due to commercial reasons rather than device limitations or technological barriers. Therefore, this research refrains from an in depth investigation of running VERSAG on such devices.

1.8 Thesis Organization

This thesis is organized into seven chapters as shown in Figure 1-5. Key publications are also shown with links to the corresponding chapters.

Chapter 2 provides a background of existing research in adaptive mobile agent research. First, an overview of compositionally adaptive software is provided, identifying the nature of adaptation which is the focus of this research. This is followed by an overview of agent mobility and migration strategies. Then, a set of criteria upon which related work has been evaluated is introduced, followed by brief descriptions of the related research work. The chapter then presents a comparative analysis of related work which elicits their strengths and limitations, and also identifies gaps in the research.

Chapter 3 introduces the proposed VERSAG agent architecture that is the main contribution of this thesis. An example scenario is first described to highlight the key attributes that a software solution for such a scenario should possess. As a solution to this scenario, we then introduce the VERSAG agent framework together with a reference architecture of an agent and definitions of key concepts. The chapter then describes how a VERSAG agent's core itinerant behaviour is implemented followed by a detailed examination of *capabilities,* which is a key concept of VERSAG. Details of the agent framework have been published in [GKL09, GKL10] and [GZK09] with early ideas/concepts reported in [GZK08a] and [GZK08b].

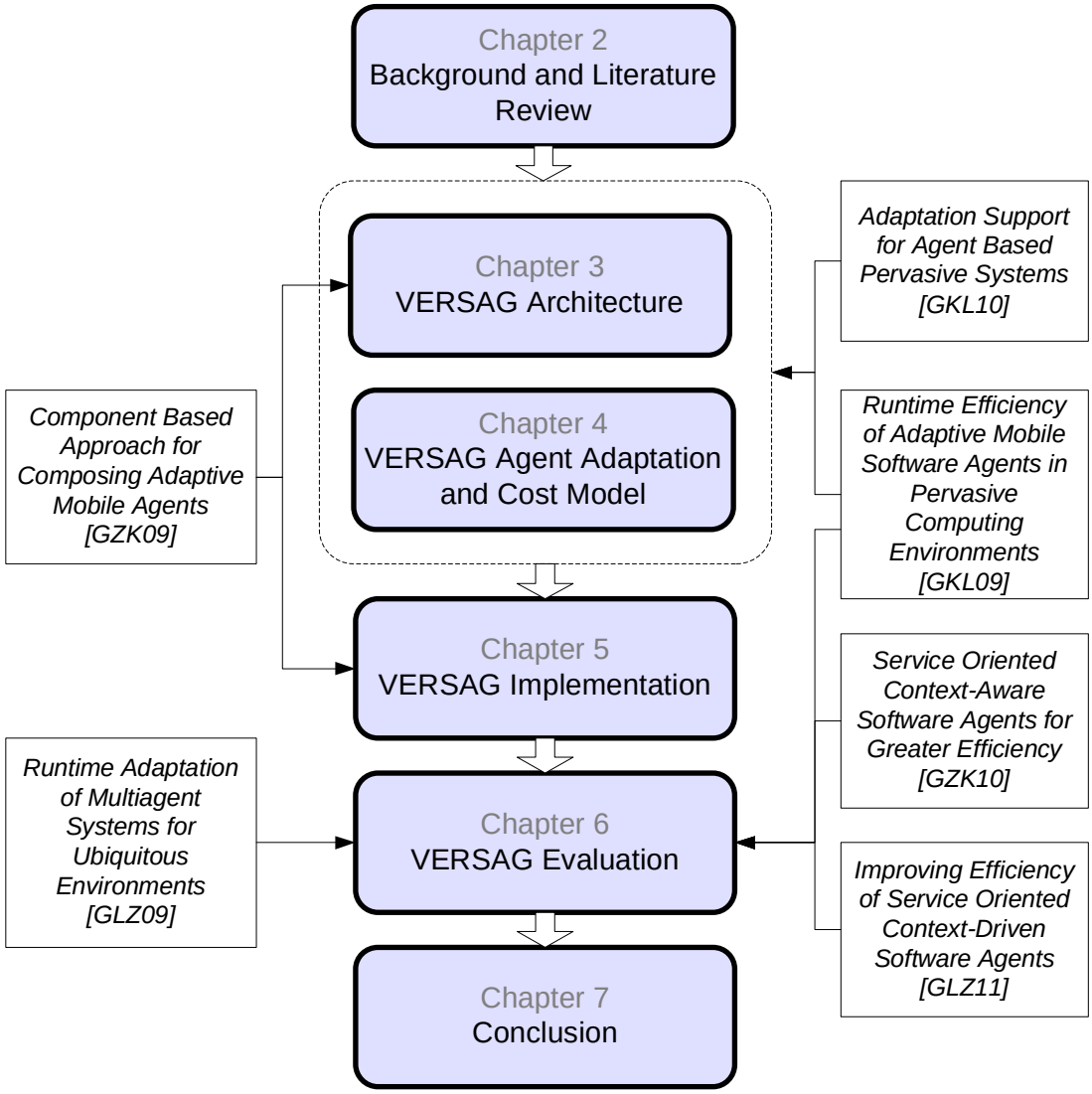


Figure 1-5: Research roadmap

Chapter 4 is dedicated to describing VERSAG agent adaptation and the decision making cost model, which enables agents to adapt cost-efficiently in dynamic environments. The chapter begins with an overview of adaptation in VERSAG agents. The second half of the chapter is dedicated to describing the multi-criteria decision making cost model, associated cost elements and mechanisms for cost estimation. Partial descriptions of the cost model have been published in [GKL09] and [GKL10] and the application of the cost model in a service oriented scenario was reported in [GZK10] and [GLZ11].

Chapter 5 discusses the implementation of the VERSAG framework. The chapter begins with a description of the tools and techniques used for the implementation.

This is followed by a description of the important decision steps involved in selecting a capability model for the prototype implementation. Design and implementation details of the prototype are described next with the aid of UML class diagrams and sequence diagrams. The final section of the chapter explains information pertaining to the operation of the prototype. Parts of this chapter have been previously published in [GKL09, GZK09] and [GKL10].

Chapter 6 presents the experimental evaluations carried out to validate the VERSAG approach. We start with a description of a case study implementation. The described experiments validate the performance benefits achievable from VERSAG, the decision making cost model and scalability of VERSAG agents. Results presented in this chapter have been published in [GKL09, GKL10, GLZ09, GZK08a] and [GLZ11].

Chapter 7 summarizes the contributions of this thesis and presents future directions for this research.

C H A P T E R T W O

2 Background and Literature Review

Chapter 1 presented background information and stated the basic premise of the current research, its aims, objectives and contributions. This chapter provides a comparative evaluation and analysis of relevant literature with regard to the research undertaken and identifies gaps in the existing body of knowledge.

2.1 Introduction

The previous chapter argued that mobile software agents can assist with important features of software for pervasive applications such as heterogeneity, mobility, context-awareness and adaptability. Recent works such as the ODDUGI mobile agent platform [CCB09], the Irish Agent Factory [MOB07] and MobiSoft [EKR08] are representative of efforts to harness this suitability of mobile agents for pervasive computing. The main aim of the current research is to harness and enhance the potential of mobile agents to build pervasive applications and services.

The previous chapter also identified combining mobility with dynamic compositional adaptation as a suitable approach to address challenges faced by software agents in pervasive environments and as the focus of this thesis. While there has been considerable research interest in adaptive agent systems [AKK03, CGG03, Gue04, MaM06, PaL05, Wei96], there has only been limited focus on compositional adaptation of individual mobile agents. The main objective of this chapter is to analyse and review the state-of-the-art in compositionally adaptive mobile software agents in the context of their suitability for pervasive computing environments.

Towards this end, we first identify a set of evaluation criteria which are indicative of support provided in the two key areas of *mobility* and *adaptation*. Then, we describe key related research in compositionally adaptive mobile agents and evaluate their support for pervasive environments based on these evaluation criteria. Using this comparative evaluation and analysis, this chapter identifies key challenges and issues faced by the state-of-the-art in supporting pervasive applications.

We begin this chapter with a further examination of the concept of compositional adaptation of software in section 2.2. Enabling technologies and a classification of compositional adaptation are provided to assist further understanding our focus. Migration strategies of mobile agents are discussed in section 2.3. In sections 2.4 and 2.5, we introduce the criteria used to evaluate the state-of-the-art and survey the existing bodies of work in this area. Section 2.6 provides a comparative evaluation and analysis of these works with respect to their suitability for pervasive applications and identifies gaps in the state-of-the-art. Finally, section 2.7 concludes the chapter with a recap of the discussions of this chapter.

2.2 Compositionally Adaptive Software

According to the Merriam Webster[2] online dictionary, to adapt is "*to make fit (as for a new use) often by modification*". Software applications in pervasive environments need to adapt to fit new usage requirements as well as to adjust to changes in their environment. McKinley et al. [MSK04b] identify two mechanisms by which software applications can be adapted: *parameter adaptation* where an application's internal variables are modified to change the program behaviour, and *compositional adaptation* where internal algorithms and structural components of the application are changed. This second approach is a stronger class of adaptation with higher impact [MSK04b, SaT09] as it allows an application to gain new strategies and algorithms in response to situations for which it was not originally designed, essentially making it "future proof". They also state that while the history of compositional adaptation could be dated back to the use of self-modifying code in

[2] http://www.merriam-webster.com

early computers, it has gained popularity in recent times mainly due to the advent of pervasive computing research.

Three key technologies which enable composing adaptive software are as follows:

i. *Separation of concerns:* This is the ability to separate the development of the functional behaviours of an application and other cross cutting concerns (e.g. quality of service, security), and is an important principle in software engineering [CMP06]. The modularity obtained from the separation of concerns enables us to implement compositional adaptation with respect to individual aspects.
ii. *Computational reflection:* A program's ability to maintain information about itself and use this information to reason about and change itself is gained through computational reflection. Reflection therefore exposes an application's internals at a sufficient level to enable compositional adaptation [MSK04a].
iii. *Component-based design:* Applications made up of software components with well-defined interfaces are promoted by component-based design. Components are reusable units of software that can be independently developed and deployed.

The same authors also propose classification schemes for compositional adaptation. One approach for grouping compositionally adaptive software is based on when the composition occurs. If the composition occurs at application development time, compilation (linking) time or at load time, it is termed as *static composition*. If the composition occurs at application runtime, it is termed *dynamic composition*. Within dynamic composition, if only the application's non-functional features can be altered, it is known as *tunable software,* while when the business logic (application functions) can also be altered, it is known as *mutable software*. The focus of this thesis is therefore on *mutable dynamic composition* of software agents.

Figure 2-1 illustrates how components can be added to and replaced from a software application at runtime. Step 1 shows a software application consisting of 4 components named *A, B, C* and *D*. In step 2, component *B* is removed and in step 3

replaced with a compatible component *B′*. Step 4 shows the modified application which is now made up of components *A, B′, C* and *D*.

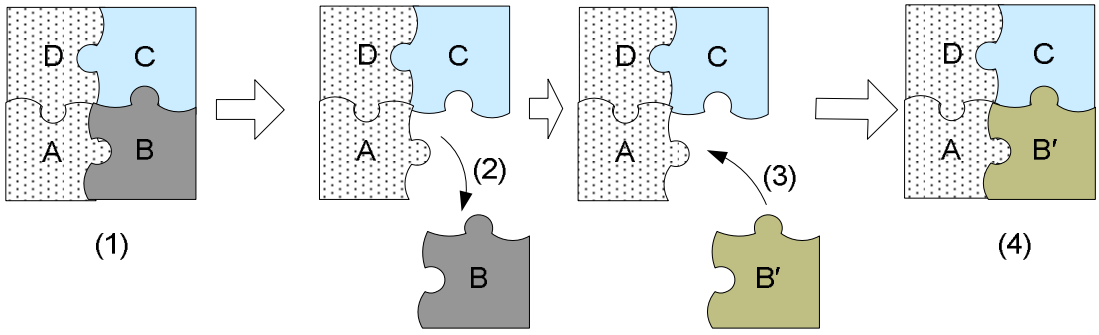


Figure 2-1: Runtime compositional adaptation illustration

Next, we provide an overview of agent mobility and agent migration strategies.

2.3 Mobile Agent Migration Strategies

As a form of code mobility [Pic01, Vig98], a key advantage that mobile agents bring to distributed computing is reduction of network bandwidth usage (see section 1.2, code shipping Vs data shipping). Despite this conceptual advantage, many mobile agent based systems suffer from poor bandwidth usage due to the simplistic migration strategies used in agent toolkits [BMK05a]. In this section we present background information on different agent migration strategies in use.

A mobile agent is generally viewed as consisting of code, data and an execution state [BrR04] and when an agent migrates, these components have to be transferred to the destination agency. However most agent toolkits, especially those based on the Java language, are not able to transfer the agent's execution state [BrR04, CPV07, Pic01, SBB00]. Hence they are considered as providing *weak mobility* while systems allowing transfer of the execution state are considered as providing *strong mobility* [BrR04, Pic01, SBB00]. Within this thesis we do not deal with strong mobility of agents since many current mobile agent toolkits are Java based and do not implement strong mobility.

A *migration strategy*, according to Braun and Rossak [page 76 of BrR04] is the approach used for moving agent code (i.e. agent's classes) as the agent migrates.

They broadly categorise migration strategies as *push* or *pull* based on whether code is pushed to the destination by the origin agency when the agent migrates or pulled by the destination agency when instantiating the agent. Further categorization is possible based on whether code is moved all at once (as a single archive) or unit by unit (e.g. class by class). This classification is shown in Figure 2-2.

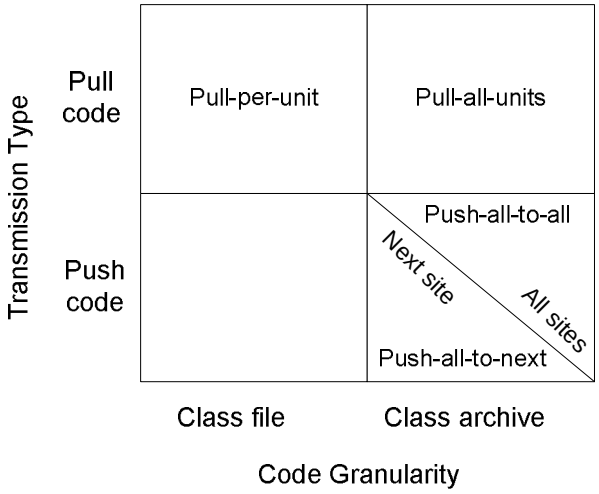


Figure 2-2: Overview of migration strategies (source [BrR04])

In *push-all* strategies, when an agent migrates, all classes belonging to it are pushed to the destination, even though some of these classes may not be used. In the *pull-per-unit* strategy, the destination pulls classes from the agent's home agency (or a remote code server) as needed (e.g. JADE [BCP03]). This results in the agent requiring continuous connectivity with a code provider as well as having higher overheads due to the use of multiple network sessions. Most agent toolkits implement simple *push-all* or *pull-as-required* migration strategies which can be inefficient. There are also approaches where only a specification of the agent is migrated (E.g. agent blueprints in the Dutch Agent Factory [SWB03]). In such cases, the destination agency rebuilds the agent based on this specification making use of components available to it.

Generally, it is not possible for mobile agents (or agent developers at design time) to make decisions on what migration strategies to use as it is typically determined by the mobile agent toolkit in use. Braun and Rossak [page 116 of BrR04] show that different migration strategies can be beneficial in different scenarios and that

therefore it is not efficient to apply a single strategy to all situations. They propose a new agent mobility model named Kalong which has three key mechanisms to improve migration efficiency of mobile agents.

1. *Adaptive transmission of code and data* lets the agent developer make dynamic decisions on which code units (and data) to send to a new agency when the agent migrates. Decisions are based on criteria such as the probability of use and network conditions. Furthermore, the agent has the ability to request the omitted code and data later if needed.
2. Agents have the ability to dynamically define remote agencies as temporary *code (and data) servers*. This enables the agent's code (and data) to be loaded from a nearby server without having to contact its home agency which may be further away. The idea of using multiple code servers to improve migration efficiency was initially proposed by Hohl et al. in [HKB97].
3. Kalong also uses detailed *code caching mechanisms* to ensure that code units (i.e. classes) that are already available at an agency are not transmitted to it again. Code units are compared for equality using a hash value and allow sharing code between different agents. Some form of code caching is implemented in most agent toolkits (e.g. JADE [BCP03], Aglets [LaO98, LOK97]).

Letting the agent decide on the strategy for each migration, dynamic setup of code servers and code caching are therefore three factors which lead to efficient agent migration. Being able to change the granularity (i.e. size) of the unit being moved is also important. For example, in a related research, Kern et al. [KBF04], achieve improved network efficiency by moving only parts of classes that are required at the destination.

Having provided relevant background information on compositional adaptation and agent mobility, in the next sections of this chapter we investigate previous work where compositional adaptation has been incorporated in mobile software agent research.

2.4 Criteria for Evaluation

In chapter 1 we established that mobile agents enhanced with compositional adaptation are a suitable approach for building smart pervasive applications. The main purpose of this chapter is to study and analyse existing works in this area in the context of their suitability for pervasive environments. In this section, we present a set of criteria for evaluating these related research works.

As described in chapter 1, the ability to adapt at runtime is an important requirement for software agents to be successful in complex dynamic environments. Hence, our first six criteria are indicative of the level of support available for agent adaptation.

i. *Self-adaptivity:* We consider an agent to be self-adaptive if it decides to adapt as a result of reasoning about its internal state and/or inputs from the environment. If an external entity carries out the adaptation of an agent, it is not considered self-adaptive. Thus, for this criterion we use a weaker notion of self-adaptive software compared to that proposed by Laddaga [Lad97, Lad00].

ii. *Adaptation scope:* This criterion is indicative of the degree of choice that an agent has when it adapts. Adaptation scope is considered to be *closed* if an agent can only use a pre-defined collection of components for restructuring itself via compositional adaptation. When no such limits exist, the scope is considered to be *open*. This criterion is similar to the *adaptation type* in Salehie and Tahvildari's [SaT09] taxonomy of self-adaptation.

iii. *Component source:* A compositionally adaptive agent faced with circumstances that it is not equipped to handle (i.e. task or environment) needs to acquire necessary components from external sources [CCD99, DPK00, FCL08]. In dynamic heterogeneous environments, the service location from where an agent can find new components is important for its success. This criterion therefore looks at what different component sources have been used in related literature.

iv. *Component reusability:* A key motivation for component based software development (CBSD) is the ability to reuse software components [CMP06, Pre01, Som11]. Since it is desirable to have a higher level of component

reusability, this criterion examines the extent to which the components used in previous works are reusable within and outside the framework they were originally proposed for.

v. *Adaptation triggers:* An agent may have to adapt in response to different triggers such as application requirements, agent migration (i.e. due to different enabling technologies of the destination device), and environmental changes (e.g. resource level changes at its current device) [CaK02, CCB09, MOB07]. The aim of this criterion is to understand the extent to which state-of-the-art agents address this concern.

vi. *Context-awareness:* As described in chapter 1, context-awareness is an important feature of pervasive applications in order to react to environmental changes. Thus, our sixth criterion is whether a considered agent system equips agents with context-awareness.

vii. *Level of mobility support:* Mobility, as identified previously, is an integral component of pervasive computing applications and is a central theme of this research. We consider three attributes as indicative of mobility support. An itinerary is a vital artefact of a mobile agent [LuX05, PKA07], useful both to model and program its mobility behaviour. Thus, the explicit *use of an itinerary* is indicative of a high level of support for mobility. As described in section 2.3, *granularity of the mobile entity* and *fine-grained control of migration* are also indicators of the level of mobility support.

Having presented our criteria for evaluation, in the next section we provide an overview of each of the related research works considered.

2.5 Related Work in Compositionally Adaptive Mobile Agent Systems

Having identified the evaluation criteria, this section presents a review of past research where mobile software agents with compositional adaptation ability have been proposed and developed. It should be noted that we have not considered research work where either of these key components are not present. For example, Malaca [AmF09] software agents and Java based Intelligent Agent Componentware

(JIAC) [HKH09] are two projects excluded since they do not have mobility as a primary concern.

A brief overview of each system is provided followed by a summary of characteristics relating to our evaluation criteria. We note that sometimes different terminology has been used in literature to describe compositional adaptation of software agents.

2.5.1 DynamiCS

Dynamically Configurable Software (DynamiCS) [TGM98, TSG01] pioneered the idea of mobile agents being containers for application specific plug-in components. The creators identify that while integrating intelligence into mobile agents is beneficial, it also makes them heavy and impede mobility. To overcome this issue they propose an architecture which allows dynamic inclusion ("plugging-in") of negotiation capabilities into mobile agents. Three design requirements underlying DynamiCS mobile agents are:

- *Role-specific functionality*. An agent carries only the capabilities needed to fulfil its current role (e.g. a role of seller, buyer).
- *On-the-fly loading*. Agent capabilities can be loaded at runtime.
- *Flexible configuration*. Agent capabilities need to be sufficiently flexible so that they can be reused in similar situations.

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 is targeted towards ecommerce applications and specifically limits itself to adapting agents to support different negotiation schemes. The plug-in mechanism in DynamiCS is implemented making use of features available from the underlying Voyager [Voy07] agent platform. Adaptation and mobility support characteristics are as follows:

- *Self-adaptivity:* DynamiCS agents are self-adaptive.
- *Adaptation scope:* Adaptation scope is unknown. However, the type of plug-ins is limited to negotiation in the ecommerce domain.

- *Component source:* There is no specific information available about the source of plug-in components.
- *Component reusability:* Plug-in components can only be used within DynamiCS.
- *Adaptation triggers:* Negotiation needs (i.e. application tasks) are the only trigger for agent adaptation.
- *Context-awareness:* Context-awareness is not present.
- *Level of mobility support:* It is unclear whether agents have itineraries and in the absence of any details on plug-in component migration, we assume an agent to be the smallest mobile unit. There is also no indication of fine-grained migration control being present.

2.5.2 Dynamic Agents

The *dynamic agent infrastructure* [CCD99] aims to provide agents with *dynamic-modifiability* of behaviour and is a complete agent architecture built using the Java programming language. Individual *dynamic agents* are general purpose carriers of programmes. An agent is composed of a fixed carrier part that provides housekeeping services and a dynamic part which contains useful capabilities. The dynamic parts are designated as either *actions* or *open servers*. The former are application specific capabilities and the latter allow new agent management services to be added to the agents, both represented as Java classes. An agent can dynamically load new programs (*actions* or *open servers*) when it is presented with a task that requires capabilities beyond what it has at present. An agent's action execution plan is represented as an *Agenda*, a scripted object which can also change at runtime. The system allows for grouping of agents into *domains* and a *coordinator* agent can be allocated to manage cooperation among agents in a domain. *Dynamic Agents* are considered suitable for "highly dynamic and distributed applications". Three representative examples given are: scheduling and tracking of a real-time manufacturing process; dynamic workflow provisioning and distributed data mining on the web. Figure 2-3 shows the structure of a dynamic agent.

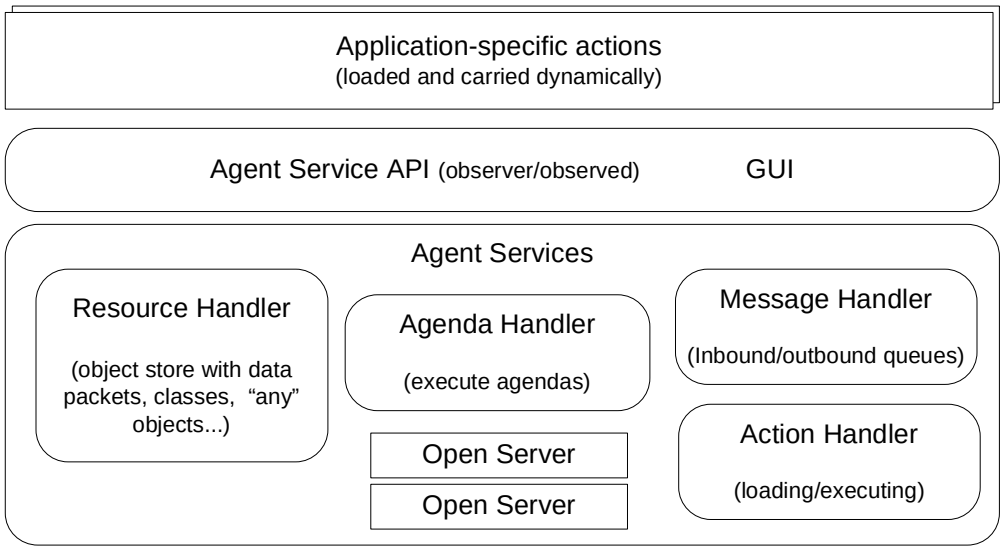


Figure 2-3: Basic structure of a dynamic agent (adapted from [CCD99])

Adaptation and mobility support characteristics are as follows:

- *Self-adaptivity:* The agents are self-adaptive and decide for themselves how and when to add new functionality.
- *Adaptation scope:* Agents have open adaptation scope as there is no limit on what functionality can be added and from where or when they can be added.
- *Component source:* An agent locates new components either from information available in request messages or by querying a resource-broker. Acquiring a component is done by downloading from a URL (Uniform Resource Locator) and adding to a local object store.
- *Component reusability:* The dynamically loadable programs need to be built according to interfaces provided which do not adhere to any standards used outside this research (i.e. inherit from `AgentAction` and `OpenServer` of the Dynamic Agent class library).
- *Adaptation triggers:* Adaptation occurs only for the purpose of carrying out tasks allocated to the agent.
- *Context-awareness:* Context-awareness is not present.
- *Level of mobility support:* An *agenda* describes what actions an agent is to perform (its action plan), however it is unclear whether it includes an agent's migration path as part of this, which would make it equivalent to an itinerary.

Mobility is supported at agent level as well as component level. There is no indication of agents having fine-grained migration control.

2.5.3 Port-Based Adaptable Agent Architecture

The Port-Based Adaptable Agent Architecture (PB3A) [DPK00] proposes an architecture for large-scale self-adaptive software based on three forms of adaptation: parametric fine tuning (i.e. parameter adaptation in [MSK04b]), algorithmic change (i.e. compositional adaptation in [MSK04b]) and code mobility. The basic building block of PB3A is a Port-Based Module (PBM) where a port is a concept to regulate communication and inter-dependency of PBMs (see Figure 2-4). PBMs can only communicate with each other through ports. Mapping of ports is handled by the *runtime core* which is the distributed Java based platform on which PB3A executes. It can change port mappings at runtime, allowing for easy reconfiguration without affecting individual PBMs. PBMs can be combined to form larger components (e.g. Macros and Port-Based Agents) with useful functionality. A Port-Based Agent (PBA) is a grouping of PBMs and is the basic unit which has self-adaptability. A PBA could consist of application-specific PBMs, performance monitoring PBMs and self-improving PBMs.

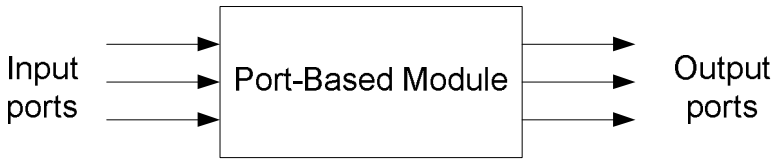


Figure 2-4: Generic diagram of a Port-Based Module (Source [DPK00])

Adaptation and mobility support characteristics are as follows:

- *Self-adaptivity:* Agents in PB3A possess self-adaptivity.
- *Adaptation scope:* The scope of adaptation is *open* as it is possible to add new PBMs to the repository dynamically and for Port-Based Agents to get new components as needed.
- *Component source:* PB3A uses a centralized repository named *module server* to store and serve port-based module code.
- *Component reusability:* Components can only be used within PB3A.

- *Adaptation triggers* and *context-awareness:* Context-awareness and discussion of adaptation triggers are absent since the architectural design of PB3A takes place at a lower level of abstraction which deals with "inputs" and "outputs". However, PB3A provides the Port-Based Driver (PBD) as a building block from which these features could be implemented. The provided example application of mobile mapping robots [DPK00] illustrates how hardware failure of a robot is monitored and triggers adaptation.
- *Level of mobility support:* PB3A does not suggest the use of an itinerary to manage agents. Being the basic building block of PB3A, the granularity for adaptation as well as migration is at the PBM level. There is no indication of fine-grained migration control being present.

2.5.4 Dynamically Adaptable Mobile Agents in Heterogeneous Environments

The work on dynamically adaptable mobile agents (DAMA) by Brandt and colleagues [BrR01] investigates how mobile agents can dynamically adapt their functionality to suit different computational environments they encounter. In the proposed approach, a mobile agent is made up of a small non-adaptable (environment independent) core, adaptation framework and adaptable parts that depend on the environment. An agent discards its environment dependent parts before migration and rebuilds itself using a suitable set of adaptable parts after arriving at the new location. To acquire environment dependent parts, a context awareness module of the agent examines the environment profile of the agent's current location and matches it with implementation profiles of various parts that are located in a repository. The authors use an illustrative example of mobile agents used for configuring a distributed set of web browsers. Environment-dependent features of importance to this illustrative task are the operating system, CPU architecture and default browser. The DAMA approach has been implemented on top of the Voyager agent toolkit [Voy07]. Adaptation and mobility support characteristics of DAMA are as follows:

- *Self-adaptivity:* Agents are self-adaptive as they carry out their own adaptation.
- *Adaptation scope:* Since repository contents can be dynamically updated with new parts, the adaptation scope in DAMA is open.
- *Component source:* There is a central repository and proxy repositories are started up closer to the agent in order to reduce network bandwidth consumption. Strategies for loading code to these proxy repositories are discussed in [Bra01].
- *Component reusability:* The reusable parts (i.e. components) are environment dependent and also match an interface which the agent expects it to have. The components of DAMA therefore have their reusability limited by environment and the interface dependencies.
- *Adaptation triggers:* Agent migration is the only considered adaptation trigger.
- *Context-awareness:* DAMA agents are context-aware.
- *Level of mobility support:* Details of itinerary support are not available while mobility granularity could be identified at component level. While there is no indication of fine-grained migration control being present, the use of proxy repositories is led by a desire to improve network usage.

2.5.5 Negotiating Agents

The negotiating agents [PAP04, PPN02] research focuses on enabling mobile agents to dynamically load reasoning models for negotiations on the Internet. A negotiating agent consists of a core that is fixed and pluggable components of three types. The three types of modules that constitute an agent are:

- *Communication module.* This module handles agent communication and is static (i.e. not pluggable) in this research.
- *Protocol module.* This module implements the negotiation protocol used by the agent. It can be dynamically loaded by the agent after figuring out the protocol needed for a particular transaction.

- *Strategy module.* This represents the reasoning strategy employed by the agent in its negotiation. The selection of a strategy is usually a secret and is dependent on the protocol as well as knowledge available about the party to negotiate with (e.g. previous negotiation history). This module is therefore also pluggable.

Figure 2-5 illustrates the components of a negotiating agent. The agent in the figure has *protocol module 1* and *strategy module 3* plugged into it while alternative modules of both types are also available to be plugged in if necessary.

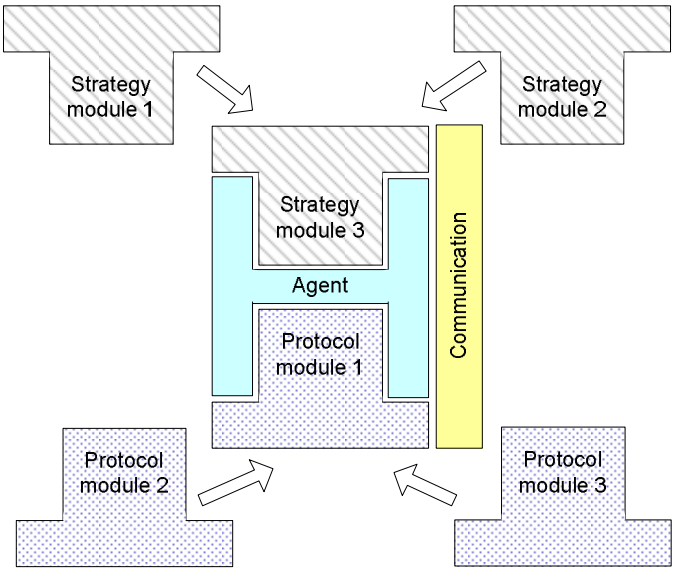


Figure 2-5: Modules of a negotiating agent (adapted from [PAP04])

Negotiating agents are implemented on top of the JADE [BCP03] agent platform. The aim of negotiating agents is similar to that of the DynamiCS [TGM98, TSG01] project and hence we note that their agents are structured similarly. Adaptation and mobility support characteristics are as follows:

- *Self-adaptivity:* Negotiating agents are self-adaptive.
- *Adaptation scope:* Adaptation scope of negotiating agents is unknown. However, type of plug-ins is limited to negotiation in the ecommerce domain.
- *Component source:* The component source is specified to be either the local machine or nearest agent server. We assume that "agent server" refers to a JADE container and that the code serving mechanism of JADE is used.

- *Component reusability:* Plug-in component reusability is limited to within negotiating agents.
- *Adaptation triggers:* Adaptation is triggered by task requirements (i.e. negotiation needs).
- *Context-awareness:* Context-awareness is not present.
- *Level of mobility support:* It is unclear whether agents have itinerary support. In the absence of any details on plug-in component migration, we assume granularity of mobility to be at agent level. There is no indication of fine-grained migration control being present. However, the advantage of having lightweight "agent-skeletons" migrate is noted as a desirable property.

2.5.6 Java Agent DEvelopment Framework

Java Agent DEvelopment Framework (JADE) [BCP03, BCG07] is a popular open source mobile agent toolkit maintained by Telecom Italia. In JADE, an agent's useful functionality is represented as programmer developed *Behaviours* which extend the `jade.core.behaviours.Behaviour` class. Support for runtime adaptation of agent functionality is provided through the `jade.core.behaviours.LoaderBehaviour` class. Through the provided APIs, a JADE agent can send a message to another agent requesting it to load a *Behaviour* dynamically. The bytecode of the classes used by the new *Behaviour* has to be sent with the request. Thus, the API provides application developers with limited control of the adaptation process.

While JADE provides the basics required for agent applications to support runtime adaptation of functionality, it is up to application developers to decide on aspects such as adaptation triggers, context-awareness, itinerary support etc. It is clear by their very nature though that these behaviours are only reusable within the JADE platform. Adaptation and mobility support characteristics are as follows:

- *Self-adaptivity:* Since the agent adapts (i.e. loads new behaviours) by itself self-adaptivity is present.
- *Adaptation scope:* With this API, it is possible for agent application developers to create applications with open adaptation scope.

- *Component source:* Since the Behaviour to be loaded is sent as part of the load request, the adaptation requestor is also the component source.
- *Component reusability:* Reuse of components (i.e. Behaviours) is only possible within the JADE platform.
- *Adaptation triggers:* While loading a Behaviour occurs as the result of receiving a request message, application level triggers are decided upon by agent application developers.
- *Context-awareness:* While JADE provides no explicit support for context-awareness, it is possible for agent application developers to create context-aware applications on top of it.
- *Level of mobility support:* Itinerary support can be added. The smallest component with mobility is a JADE Behaviour, and fine-grained migration control is not possible.

2.5.7 Dutch Agent Factory

The Dutch agent factory [BOS02, SWB03, SWB04] is part of the DESIRE (DEsign and Specification of Interacting REasoning components) [BJT02] compositional multi-agent design methodology. It provides an architecture for structuring agents so that their adaptation is easier. Unlike many systems where the agent itself adapts, here the adaptation is carried out by an agent factory external to the agent. Also, who identifies the need for adaptation is considered to be outside their research focus. The agent factory uses components locally available with it for adaptation. The proposed agent structure is component-based and identifies the following two constituents of an agent.

- *Components* are the active part (e.g. processes) of an agent.
- *Data types* represent the passive parts (e.g. classes).

A component has an interface describing its input/output data types. A component also has *slots* into which other components or data types can fit in. Slots are regulated and limit what can be included in them. An agent itself is a component made up of multiple slots. Other components implementing the agent's basic features can be connected to these slots. For example, the *generic agent model* [BJT02]

component has slots for managing the agent's internal processes, managing communications and so on. A BDI agent [RaG95] would have the internal process management slot filled with a component that handles belief-desire-intention commitments. The *application specific task* slot of an agent would be filled with a component that can fulfil the tasks allocated to the agent. The agent factory also proposes the use of an agent *blueprint* instead of code for agent migration [BOS02]. The destination platform would then reconstruct the agent based on the blueprint using components available to it. This approach has advantages for migrating agents over heterogeneous platforms as well as for improved trustworthiness of migrating agents.

Adaptation and mobility support characteristics of the Dutch agent factory are as follows:

- *Self-adaptivity:* Agents of this approach are not self-adaptive since an agent factory external to the agent carries out adaptation.
- *Adaptation scope:* Adaptation scope is closed since only components co-located with the agent factory can be used.
- *Component source:* The agent factory can only use components physically co-located with it as there is no component repository [SWB04].
- *Component reusability:* Component reusability is limited to the agent factory approach. The research does not focus on the format of the reusable parts.
- *Adaptation triggers:* Adaptation is triggered as a result of an external request.
- *Context-awareness:* Context-awareness is not present.
- *Level of mobility support:* Itinerary support is not evident in the Dutch agent factory while mobility is at agent level. Also, there is no indication of fine-grained migration control being present.

2.5.8 Generic Adaptive Mobile Agent Architecture

The Generic Adaptive Mobile Agent architecture (GAMA) [AmF05] takes a component-based approach to developing adaptive mobile agents. Like dynamic agents [CCD99], GAMA also is a complete agent architecture built from ground up using the Java language. A GAMA agent is composed of multiple components (e.g. a

kernel, an agent interface, a mobility service, communication service, knowledge base, application specific functions etc.) as shown in Figure 2-6 and is capable of adapting itself to suit new environments encountered as the agent migrates.

Adaptation in GAMA is seen as a process which takes as input, a core environment independent agent, a set of components which depend on the environment and a description of the current environment, and produces as output the core agent linked with a selected set of components that suit the environment. Two of the agent's components provide it with context-awareness and self-adaptation ability. The restructuring of the agent is done by loading components from a repository which is available locally at its current location. The purpose of adaptation is to be able to execute when faced with environmental changes, yet agents are not responsive to environmental changes which occur while they are stationary (e.g. variations in CPU load, battery power etc.).

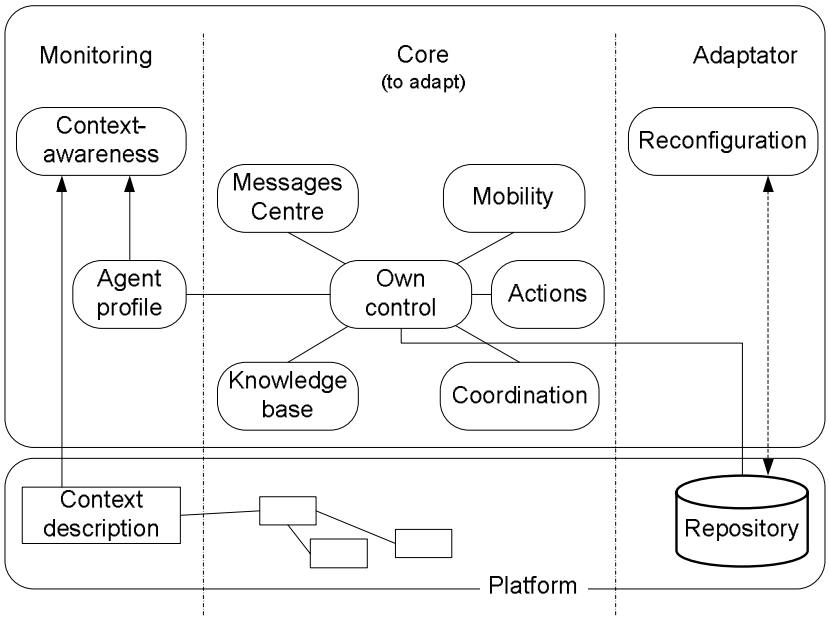


Figure 2-6: GAMA self-adaptive mobile agent architecture (adapted from [Ama06])

Adaptation and mobility support characteristics of GAMA are as follows:

- *Self-adaptivity:* GAMA agents are self-adaptive.
- *Adaptation scope:* Agents have closed scope of adaptation because selection is limited to the set of components available at its new location.

- *Component source:* The component source is a repository located at the agent's current location.
- *Component reusability:* Components can only be reused within the GAMA architecture.
- *Adaptation triggers:* Adaptation is triggered as a result of agent migration.
- *Context-awareness:* Context-awareness is present in GAMA agents.
- *Level of mobility support:* GAMA agents can be equipped with an itinerary while mobility is only available at agent level. There is no indication of fine-grained migration control being present.

2.5.9 Self-Configuring Personal Agent Platform

In [FCL08], Feng et al. describe an agent platform that consists of self-configuring agents and is targeted towards creating personal assistant agents in pervasive computing environments. We identify this platform as SCPA (Self-Configuring Personal Agents). It consists of the following four major components.

- *Ubiquitous Intelligent Objects (UIO)*. UIOs are used to represent human users and other physical objects that deal with the agent system. Each UIO has an associated sensory tag to identify it. Details of UIOs, such as associated attributes, capabilities and provided services are stored by the system in its databases.
- *Personal agents*. Each user in SCPA has an associated personal agent to represent him or her in the virtual environment. An agent is also represented as a UIO in the system.
- *Code repository*. The code repository is the centralized store of executable and mobile code in SCPA. Functional descriptions and details of required parameters to execute this code are also stored together. Mobile agents dynamically request code from the repository over HTTP (hypertext transfer protocol).
- *Service registry*. UIOs can publish the services they provide in the central service registry of SCPA.

A personal agent in SCPA is made up of eight components as shown in Figure 2-7. The *Profile* maintains details such as user preferences. The *Monitoring*, *Analysis* and *Planning* components together let the agent sense its context, deliberate on the sensed context information and plan the agent's actions. The *CodeDownloading*, *ParameterFormFilling* and *Execution* components collaborate to acquire any necessary mobile code and parameters and then run the code as per the planned actions.

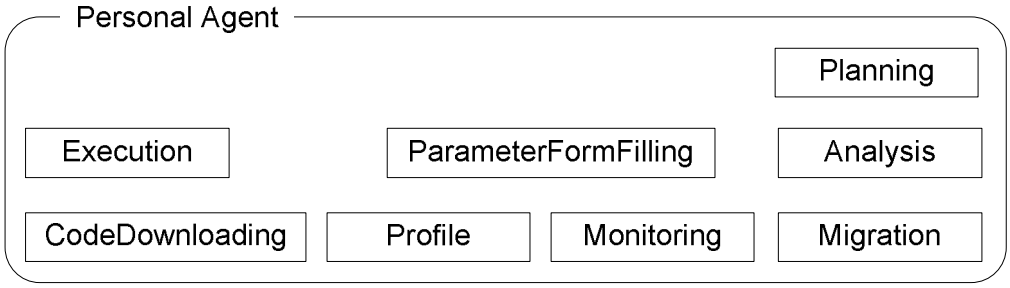


Figure 2-7: Structure of self-configuring personal agent (adapted from [FCL08])

In SCPA, code required to interact with UIOs are developed by the UIO suppliers and added to the code repository, from which personal agents can dynamically acquire code. The code is associated with a description of functionality provided as well as parameters required to execute it. Further details of any restrictions or standards adhered to by the reusable code is not available.

It should be noted that the SCPA research work is parallel to our research presented in this thesis. Adaptation and mobility support characteristics of SCPA are as follows:

- *Self-adaptivity:* SCPA agents are self-adaptive.
- *Adaptation scope:* Adaptation scope is open as new code can be dynamically added to the repository and subsequently used by agents.
- *Component source:* A centralized code repository accessible over HTTP is the source of components in SCPA.
- *Component reusability:* Components are built only for use within the current platform.

- *Adaptation triggers:* Adaptation can start as a result of task requirements or due to sensed contextual changes.
- *Context-awareness:* SCPA agents are context-aware.
- *Level of mobility support:* There are no details of itinerary support in SCPA. Since components can be downloaded from a remote repository, mobility granularity is at component level. There is no indication of fine-grained migration control being present.

Having described the related literature, the next section presents our analysis and summary comparative evaluation of the state-of-the-art of compositionally adaptive mobile software agents.

2.6 Analysis of Compositionally Adaptive Mobile Agent Systems

This section provides a comparative evaluation and analysis of the related research previously described, based on the criteria outlined in section 2.5. The analysis is carried out for the purpose of identifying their suitability for building pervasive applications, and areas in which improvements are needed. A comparison of the related work is summarised in Table 2-1. The significant observations made are as follows:

- Self-adaptivity of agents is available in all the considered systems except the Dutch Agent Factory [SWB03] where the main research focus is the use of an external agent factory to adapt agents.
- Adaptation scope is *open* in most systems with only GAMA [AmF05] and the Dutch Agent Factory [SWB03] clearly identifiable as having *closed* scope.
- The component source of choice is a remotely located code repository accessible over the network. Once again, only GAMA [AmF05] and the Dutch Agent Factory [SWB03] require the components to be co-located with the agent. DAMA [BrR01] creates proxies of its central repository closer to the agent in order to reduce network bandwidth consumption in component acquisition.

- While component reusability is a major advantage of all these systems, each system defines its own interfaces for components. The possibility of reusing these components outside their respective frameworks has not been investigated.
- The early systems only consider agent adaptation as being triggered by application tasks for which the agent does not possess required components/code (e.g. DynamiCS [TGM98, TSG01], Dynamic Agents [CCD99] and Negotiating agents [PAP04, PPN02]). However, more recent systems identify migration and environmental changes as triggers for adaptation. Similarly, the more recent works see agents being context-aware (GAMA [AmF05] and SCPA [FCL08]). These trends are indicative of the growing focus on the heterogeneity of computing environments that the agents execute in and the necessity of agents being context-aware in such environments.
- Support for mobility is mixed with only GAMA [AmF05] clearly identifying the use of an agent itinerary. Mobility granularity is seen at both agent and component level while none of the systems appear to support efficient agent migration through fine-grained migration control.

Table 2-1: Comparative evaluation of related work

	Self-adaptive	**Adaptation scope**	**Component source**	**Component reusability**
DynamiCS (1998)	Yes	Not specified	Not specified	Within
Dynamic Agents (1999)	Yes	Open	URL	Within
PB3A (2000)	Yes	Open	Remote repository	Within
DAMA (2001)	Yes	Open	Repository and proxies	Within
Negotiating Agents (2002)	Yes	Not specified	Agent server	Within
JADE (2003)	Yes	Open scope possible	Requesting entity	Within
Dutch Agent Factory (2003)	No	Closed	Local computer	Within
GAMA (2005)	Yes	Closed	Local repository	Within
SCPA (2008)	Yes	Open	Remote repository	Within

Table 2-1: Comparative evaluation of related work (continued)

	Adaptation Triggers	**Context aware**	**Level of Mobility Support**		
			Itinerary	**Granularity**	**Fine-grained control**
DynamiCS (1998)	Task	No	Not specified	Agent	Not specified
Dynamic Agents (1999)	Task	No	Not specified	Component (action program)	Not specified
PB3A (2000)	Not applicable	Possible	Not specified	Component (PBM)	Not specified
DAMA (2001)	Migration	Yes	Not specified	Component	Not specified
Negotiating Agents (2002)	Task	No	Not specified	Agent	Not specified
JADE (2003)	Not applicable	Possible	Possible	Behaviour	No
Dutch Agent Factory (2003)	External request	No	Not specified	Agent	Not specified
GAMA (2005)	Migration	Yes	Yes	Agent	Not specified
SCPA (2008)	Task and context	Yes	Not specified	Component	Not specified

Based on the above comparative evaluation, we recognize that the state-of-the-art in compositionally adaptive mobile software agent research lacks support for pervasive computing in the following key areas.

i. *Dependence on a single component source*: A challenge faced by software agents in pervasive environments is connectivity variations that range from high speed reliable links to slow wireless links with poor reliability and intermittent loss of connectivity [NiL04, SSP04]. In such dynamic environments, an agent's dependence on a single remote repository for required software components can lead to poor performance as well as task failure. For example, *dynamic agents* [CCD99], always download components from a remote URL, and multiple co-located *dynamic agents* needing the same component would end up separately downloading the component. This is a weakness that all related literature examined suffer from except DAMA [BrR01], which attempts to overcome it with the use of proxy repositories.

ii. *Fragmented view of adaptation*: Current systems view agent adaptation as being 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). The literature examined above usually considers only one or two of these as adaptation triggers. However, all these are valid needs for adaptation and it is necessary for agents in pervasive environments to possess an integrated view which takes into account all of these adaptation triggers.

iii. *Limited mobility support*: The systems investigated do not pay particular attention to agent mobility. In fact none of them are seen to employ mechanisms to improve migration efficiency and most even do not have itinerary support which is a key feature of mobile agent systems. With mobility being an important aspect of pervasive applications and poor migration efficiency a common issue in mobile agents, we identify limited support for mobility as a weakness that needs to be addressed if one is to realise the potential of mobile agents for pervasive computing.

iv. *Limited component reusability:* Another observation we make is that software components used by these agents can only be reused within that particular framework. If their scope of reuse can be expanded beyond one agent framework, this would be a step towards making them into commercial-off-the-shelf (COTS) components [CMP06, Pre01]. Adhering to widely accepted standards when building agent components is a clear mechanism to achieve this.

We also observed that software agents executing in pervasive environments need to acquire new functionality in order to interact with previously unknown hardware and software entities (e.g. to migrate to and execute on a new type of device) [AmF05, BrR01]. In such cases, it should be possible for agents to search for and locate new functionality as needed. In other words, agents should have *open adaptation scope* as observed in most of the above literature.

Based on our analysis, it can be clearly seen that the state-of-the-art in compositionally adaptive mobile software agents lacks support for pervasive computing applications in terms of both adaptation and mobility features. Therefore, the aim of this thesis is to develop a mobile agent framework to enable building pervasive applications, which have features of dynamic compositional adaptation and cost-efficient and fine-grained mobility.

2.7 Summary

This chapter reviewed and analysed the state-of-the-art in compositionally adaptive mobile software agents. The first part of the chapter presented background information on compositionally adaptive software and mobile agent migration strategies.

In the second part of the chapter we introduced and discussed the state-of-the-art in related research. The analysis, which focused on investigating the suitability of existing systems for pervasive application development, was based on seven evaluation criteria selected to elicit their support for adaptation and mobility. These were: self-adaptivity, adaptation scope, component source, component reusability, adaptation triggers, context-awareness and level of support for mobility.

The comparative evaluation and analysis illustrated that current systems do not make maximum use of the advantages of compositional adaptation and lack support for pervasive environments. Specific open issues and challenges identified in the existing literature are:

i. Dependence on a single component source
ii. Fragmented view of agent adaptation
iii. Lack of focus and support for mobility and
iv. Limited scope of reuse for components

Therefore, in the remainder of this thesis we propose, develop and validate a novel agent based framework as a solution towards overcoming the weaknesses found in the state-of-the-art. We specifically focus on the first three challenges and aim to build an approach in which agents are not dependent on a single component source, have an integrated view of agent adaptation and have a high level of mobility support. The next chapter introduces this proposed solution with the aid of a motivating scenario and further describes its architecture and underlying formal concepts.

C H A P T E R T H R E E

3 VERsatile Self-Adaptive AGent Architecture

In the preceding chapter, we provided a comparative evaluation and analysis of relevant literature in compositionally adaptive mobile agents and identified gaps in terms of their suitability for pervasive computing. This chapter introduces our proposed VERSAG agent framework and details its components and underlying concepts.

3.1 Introduction

As discussed in chapter one, mobile software agents are a suitable approach to address the challenges of pervasive application/service development due to their inherent adaptability. It was also noted that there is an increased interest in building adaptive agent systems to cope with changing requirements in dynamic heterogeneous environments. This interest has been partly fuelled by developments in pervasive computing where dynamic requirement changes and heterogeneous environments are the norm. In this context, the previous chapter discussed the state-of-the-art in compositionally adaptive mobile software agents. It was seen that existing adaptive mobile agent systems lack certain features of compositional adaptation necessary for pervasive environments. It was specifically noted that existing systems suffer from the following weaknesses:

i. *Dependence on a single component source.* In pervasive environments, this can lead to poor performance in low bandwidth networks and also to task failure due to unforeseen network disconnections.

ii. *Fragmented view of adaptation.* Agent adaptation is typically necessitated by a range of reasons including functional requirements, agent migration and environmental changes. Current systems lack an integrated view of adaptation and as a result are unable to adapt in response to *all* of these triggers. Thus, they are restricted to adaptation driven by one or two of these reasons.

iii. *Limited mobility support*. Existing systems contain pre-defined agent migration techniques which are inefficient in pervasive scenarios where environmental conditions (e.g. network parameters) vary. It is desirable to have a flexible approach to migration that allows it also to be adapted according to needs.

iv. *Limited component reusability*. Software components used by these systems cannot be reused beyond the frameworks for which they were originally designed for. It is beneficial to expand their scope of reuse to other environments by adhering to widely accepted standards.

Thus, we have identified that there is an opportunity to leverage mobile agent technology to provide new solutions targeted specifically for pervasive environments. This chapter introduces, as the main contribution of this thesis, a versatile enabling technology for building dynamic pervasive applications and services through the use of compositionally adaptive mobile software agents. This proposed novel agent framework is named VERSAG (VErsatile Self-adaptive AGents).

We begin the chapter with a hypothetical scenario in section 3.2 which motivates the need for compositionally adaptive mobile software agents. We use this scenario throughout the chapter to illustrate various concepts as they are introduced. In section 3.3, the VERSAG framework is introduced, its key features identified, a conceptual architecture presented and main concepts defined. Section 3.4 describes how the itinerant behaviour of an agent is implemented in the agent kernel. Capabilities are a key feature of VERSAG agents that underpin the framework adaptation strategy.

Therefore, a detailed examination of various aspects of VERSAG capabilities is presented in section 3.5. In this section, we identify requirements of a capability model, describe the life cycle of a capability inside a VERSAG agent, describe the peer capability sharing feature, and look at issues related to discovering and describing capabilities. The chapter is concluded with a summary in section 3.6.

3.2 Motivating Scenario for Compositionally Adaptive Mobile Agents

To motivate and illustrate the use of the proposed agent framework in a pervasive application scenario, the first hypothetical example introduced in section 1.5 is described in detail here.

Bob has just arrived at his office and needs to upload a memo he typed on his smart phone while on the way to work to the office virtual noticeboard. He also needs to search for documents in the office Intranet which refer to their m-commerce product, so that they can be updated to reflect the latest changes in the product. Bob would like to access this list of documents from his laptop computer, which is his primary working device at office.

A non-agent based solution to automate these tasks requires, in the first instance, the smart phone to have installed on it a client application which can access the memo on the phone and upload it to the virtual noticeboard, and in the second instance, a client application which can search different types of documents and servers on the office Intranet. While the first task may be relatively simple, the second task involves accessing multiple services (e.g. file servers, databases, shared Wikis, internal web sites) to search different types of files. The client interface for this query is likely to be fairly complex and possibly not suited for access from a small device such as a smart phone. This approach also has the disadvantage that a different client application is needed for each task. As such the problem lends itself naturally to the use of mobile agents. We now examine several variations of agent based solutions.

One option is to use several stationary agents, with a personal assistant agent residing on the smart phone accepting the user's requests and using the assistance of task-

specific agents to carry them out. First, the services of a virtual noticeboard client agent would be used by passing to it the memo contents and Bob's authentication details. The document search task could be delegated to one or several information retrieval agents that reside in the office network. Finally, the results have to be passed on to another agent that resides on Bob's laptop computer. While more suited for the tasks at hand than the non-agent solution, this approach too has several disadvantages as follows.

- Since the coordinating (personal assistant) agent resides on a mobile device, network disconnections and resource restrictions can affect task coordination.
- It requires complex communication and coordination between multiple service agents.
- User's sensitive personal data (e.g. authentication details) have to be shared with external agents.
- A separate agent is needed simply to display the results on a different device.

It is possible, by making the personal assistant agent mobile, to have it migrate to the office network during the task execution period and thereby mitigate the risks due to network disconnections and resource restrictions on the mobile device. If the personal assistant agent contains the necessary functionality, it could fulfil the final task of displaying results too. However, this would result in the agent needing a separate user interface for the laptop computer, which would make it larger and potentially too heavy to execute on the smart phone.

Following the same arguments, a single non-adaptive mobile agent with all the required functionality coded into it is likely to be too heavy for execution on the resource constrained mobile device. It would also be fairly large and not suited for migration over the wireless network. This approach is also inflexible since it limits the abilities of the personal assistant agent to what can be foreseen at design time.

A more suitable option is to use a mobile agent which can adapt its functionality at runtime based on task requirements and environmental conditions. The agent would first convert the user request to an itinerary which requires it to migrate from the smart phone to a computer on the office network, carry out the necessary tasks and

then migrate to the laptop computer with the results. This itinerary, as illustrated in Figure 3-1, requires the agent to migrate twice over the office wireless network. The agent would first migrate from the smart phone to the office network carrying with it the itinerary and memo contents, but no additional capabilities so as to minimize its size when traversing the wireless link. Once on the office network, it can acquire the capabilities needed to update the virtual noticeboard from peer agents. It should be noted that the agent already contains the user's authentication details and does not need to expose them to other agents. Then, the agent would ask peer agents for capabilities which would let it search the office Intranet and use them to carry out the search. Once the search is complete, before migrating to the laptop computer, the agent can discard the capabilities it no longer needs since migration is once again over a wireless link. Once on the laptop computer, the agent can present Bob with a more sophisticated user interface instead of the lightweight and simple interface it had on the smart phone. In this approach, if Bob had been called into an unexpected meeting, he can request to the agent to show him a summary of the results on his smart phone instead of a detailed report on the laptop computer. The agent achieves this by dynamically altering its itinerary to reflect the new requirements.

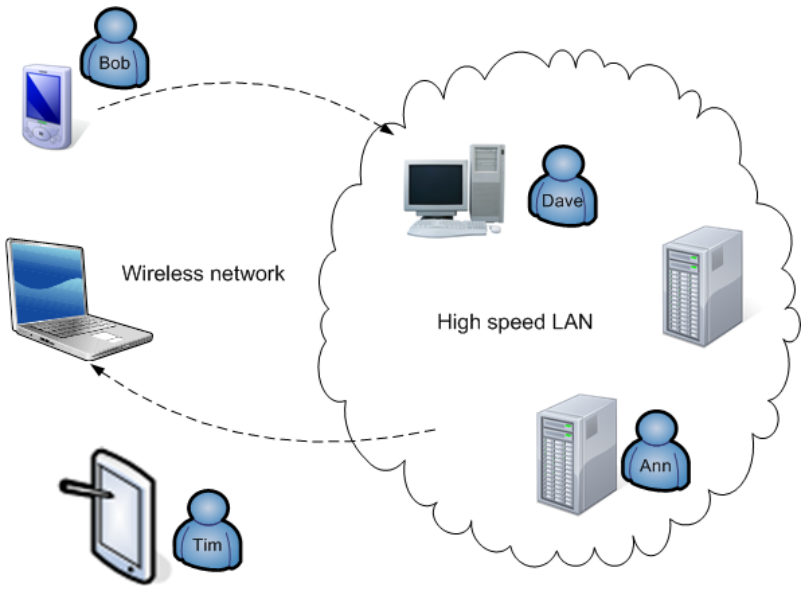


Figure 3-1: Scenario showing personal assistant agents in office network

This solution is elegant in that a single agent representing the user carries out all the required tasks, and has several advantages over the other approaches. First,

interaction with other agents is only to share functionality (i.e. code) and does not involve sharing sensitive information. The solution is also generic in that it can be adapted to meet different application needs by supplying agents with different capabilities. Network bandwidth usage can also be made more efficient by agents 'travelling light' over slow or expensive network links and acquiring capabilities from nearby peers after migration [GZK08a].

We note below features that an agent should possess in order to support this solution.

- An agent should be able to search for, acquire and use software components implementing required functionality from neighbouring agents.
- When several alternatives are available, the agent should be able to choose components that best meet user and application needs.
- An agent should be able to adapt to suit execution on heterogeneous devices including resource-constrained mobile devices. This indicates the need to be context-aware in order to use environmental conditions to guide its activities.
- It is also desirable that an agent is able to convert a high-level user request to a detailed step-wise itinerary listing locations to visit and components to execute at each location.

This research proposes an agent framework where these features are gained through a component based agent architecture with features of component sharing between agents, context-awareness and adaptability. The next section introduces the conceptual architecture of our proposed novel agent framework.

3.3 VERSAG Conceptual Architecture

In this thesis, we propose develop and evaluate the VERsatile Self-adaptive AGents (VERSAG) framework which is centred on a component-based agent model that allows agents to dynamically share components with peer agents and to adapt based on contextual needs. A VERSAG agent's application-specific functionality is provided in the form of reusable software components termed *capabilities*[3] and this

[3] We use the term capability and component interchangeably within this thesis.

facilitates an agent to gain diverse behaviours by using appropriate capabilities. Thus, 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, search for and acquire capabilities from external sources, execute capabilities it possesses, discard unwanted capabilities and terminate itself. An itinerary [PKA07] is used to define its migration path and activities. Agent adaptation 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, it is able to search for, select and acquire them from peer agents who are willing to share their capabilities as well as a centralised repository containing capabilities.

The six primitive operations and the peer capability sharing feature form the nucleus of VERSAG. 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 allows building complex and sophisticated agents which are also highly versatile. Figure 3-2 presents an overview of these features of VERSAG.

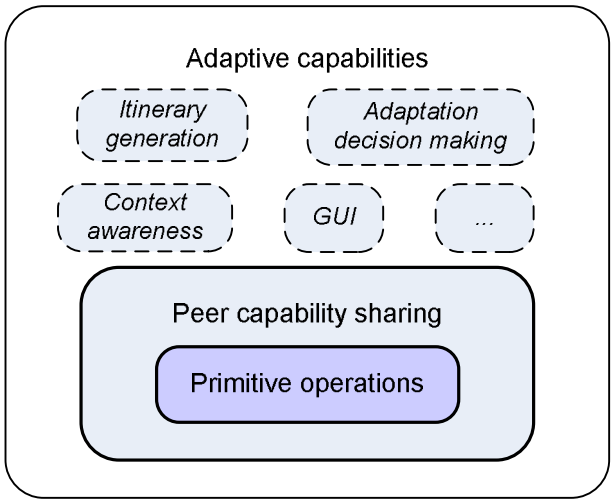


Figure 3-2: Overview of VERSAG agent features

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 functions such as reasoning ability, context-awareness and capability sharing protocols. For example, an agent can acquire a capability which implements a reinforcement learning [KLM96] strategy 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 therefore 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.

We argue that these features make VERSAG agents especially suited for pervasive computing scenarios where heterogeneous environments and rapidly changing requirements are commonplace. They are further elaborated in the remaining sections of this chapter and in the remainder of this thesis. First, in the next sub-section we present the reference architecture of a VERSAG agent.

3.3.1 Agent Reference Architecture

This sub-section presents the conceptual architecture of a VERSAG agent. In Figure 3-3, the core modules which make up a VERSAG agent are shown in a darker shade while auxiliary services are shown in a lighter shade with dashed lines. We describe each of the modules below.

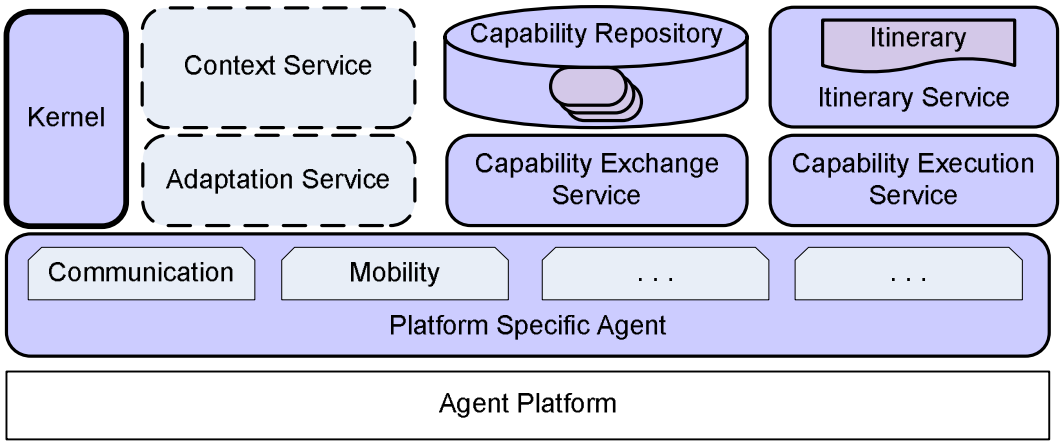


Figure 3-3: Reference Architecture of a VERSAG Agent

- *Platform specific agent:* It is expected that VERSAG agents will be used to build applications on top of existing agent platforms (e.g. JADE [BCP03], Voyager [Voy07]). 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 the current research, 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.
- *Kernel:* The kernel is the agent's main controller and its primary responsibility is executing the agent's itinerary. Thus, the kernel implements the itinerant behaviour of the agent. To execute an itinerary, the kernel utilizes primitive operations which allow it to migrate, search for and acquire capabilities from external sources, execute capabilities it possesses, discard unwanted capabilities and terminate itself. The kernel makes use of other modules for its functioning and passes control to them when required. Further details of the kernel module are given in section 3.4.
- *Capability Repository:* This is the agent's personal storage space where its own and acquired capabilities are kept. The repository 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. 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.
- *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. Its main purpose is to abstract the itinerary format and

present other modules (and capabilities) with a standard interface which remains unchanged in the event of changes to the itinerary format.

- *Capability Execution Service:* This service focuses on executing capabilities that are available in the agent's repository. Thus, it 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. Requirements of the capability execution service are presented in sub-section 3.5.1 and its implementation discussed in chapter 5.
- *Capability Exchange Service:* This service provides the key feature of capability sharing between peer agents. It fulfils 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. It is expected that in most situations this service would run as a low priority process secondary to the agent's main tasks, and may even be disabled when not required. The protocols used for capability exchange can be changed by replacing this module.

Two auxiliary modules of an agent are as follows.

- *Adaptation Service:* This 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. It is possible, in scenarios where agent adaptation is not required, for the module to be removed from an agent.
- *Context Service:* This is another auxiliary service 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.

We now take a brief look at how Bob's personal assistant agent from section 3.2 would perform its tasks if it were implemented using the VERSAG agent framework. Figure 3-4 illustrates the main steps involved in the first task of uploading the memo.

On the smartphone location, the agent needs at minimum a lightweight memo typing capability (*Editor*), a context sensing capability and a user interface capability (*GUI*). Once Bob has assigned the tasks to the agent, the agent senses, via a context sensing capability, that its current host device is resource constrained and not suitable for carrying out the requested tasks.

Step 1: Sensing that migration has to be over a wireless link, the agent first discards the *GUI* and *Editor* capabilities which are no longer needed.

Step 2: The agent migrates to a more powerful computer on the office computer network.

Step 3: Once on the office network, the agent first searches for and acquires a capability which lets it act as a client of the virtual noticeboard.

Step 4: The agent uploads Bob's memo to the virtual noticeboard using the acquired client capability.

Once the task is completed, this client capability too can be discarded. For the next task, the agent needs to first acquire a capability which can identify the various document sources to be searched and to coordinate the task. It also needs to acquire client capabilities which can communicate with these document sources. Once the search is completed, the agent can discard these capabilities while retaining the search results. Finally, the agent acquires a suitable user interface capability to run on Bob's laptop computer and migrates onto it.

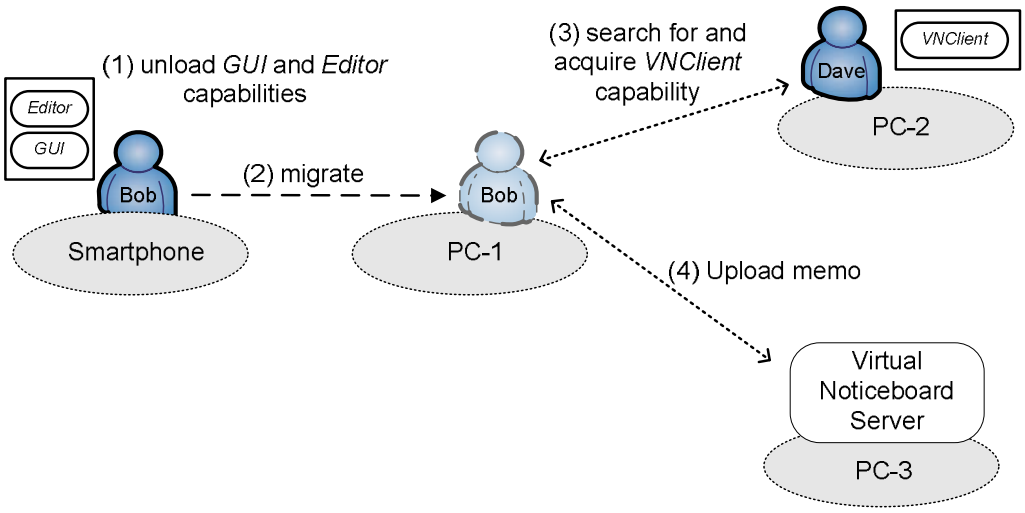


Figure 3-4: Bob's personal assistant agent implemented using VERSAG

Having provided an overview of our proposed agent framework, we now define and describe several concepts which are central to VERSAG.

3.3.2 Key Concepts of VERSAG

The key concepts of the VERSAG agent approach are formally defined in this sub-section. These definitions are used subsequently to illustrate agent behavioural logic.

Operation

A VERSAG agent supports six primitive operations which form the basis of all its activities. The operations are grouped into two subsets representing operations which are applied on agents and operations which are applied on capabilities.

Definition 1:

A primitive operation is represented as o_i, where $1 \leq i \leq 6$ and $o_i \in O$, the set of primitive operations. O is made up of two subsets O^a and O^c where,

$O^a = \{move, terminate\}$ denotes agent operations and
$O^c = \{get, discard, start, stop\}$ denotes capability operations

Each operation can be described as follows:

- $move$: request the agent to move to a specified location

- $terminate$: request the agent to self destruct
- get: search for and acquire a specified capability to the agent's repository
- $discard$: if the specified capability is in the agent's repository, remove it
- $start$: execute the specified capability if it is in the agent's repository and is not already running
- $stop$: if the specified capability is running, stop it

Applying an operation o_i on an $entity_j$ is denoted by $o_i(entity_j)$ where $entity_j$ can be either a capability or a VERSAG agent depending on the type of operation.

Function

Application-specific and other agent behaviours beyond the primitive operations in VERSAG are encapsulated in software components termed *capabilities*, and instructions given to agents have to be ultimately specified in terms of capability executions. Thus, it is necessary to be able to unambiguously specify the functionality implemented by a capability in order to identify which capabilities can fulfil a given task. We use the concept of a VERSAG *function* to indicate an atomic task that a capability implements. A capability may implement one or more functions.

Definition 2:

A *function* f_i is denoted as a pair (f_i^{desc}, f_i^{exec}) where f_i^{desc} denotes the function description and f_i^{exec} denotes an executable that encapsulates the function. The set of all possible functions is represented as $\boldsymbol{\mathcal{F}} = \{f_1, f_2, \dots, f_n\}$ where $n \in \mathbb{Z}^+$.

Representing such software functionality, as in a VERSAG *function*, in a manner that supports automated reasoning is an active area of research [PTD08, SWK02]. We discuss some of these in sub-section 3.5.5. We represent a function with a textual description and the name of the executable (a Java Archive in this case) as follows:

$$f = \langle \text{"jdbc driver for mysql"}, \text{monash.mysql11.jar} \rangle$$

Capability

A capability is a central concept in VERSAG and encapsulates functional behaviour that an agent can use to fulfil its itinerary obligations and to enhance its sophistication. It needs to be uniquely identifiable and should describe contained VERSAG *functions* and required execution environments.

Definition 3:

A capability c is represented as a 5-tuple (u_i, F_c, E_c, t_i, M) where $u_i \in U$, $F_c \subset \boldsymbol{\mathcal{F}}$, $t_i \in T$ and $E_c \subset E$. Here,

U denotes a set of unique identifiers, where $U = \{u_1, u_2, \dots, u_m\}$,
$\boldsymbol{\mathcal{F}}$ denotes the set of all possible functions $\{f_1, f_2, \dots, f_n\}$ as in Definition 2,
$E = \{e_1, e_2, \dots, e_p\}$ denotes all possible execution environments,
$T = \{oneshot, cyclic, passive\}$ denotes the execution type,
M represents meta-data about the capability, and $m, n, p \in \mathbb{Z}^+$.

Hardware and software requirements for capability execution are specified under its required *execution environment*. This could refer to specific versions of software runtimes (e.g. Java Runtime Environment) and input/output needs (e.g. keyboard, display, touch screen). Meta-data of a capability includes additional details such as dependencies on other capabilities, ownership, software version, security certificates, licensing details, supported optimizations and details of algorithms used.

VERSAG capabilities are grouped into three execution types based on their runtime behaviour. They are:

- $oneshot$: 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 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.

An illustrative capability specification which provides database driver functionality for accessing a MySQL database is shown in Figure 3-5. Here, the identifier is alphanumeric while execution environment refers to a specific Java Runtime Environment version.

$$id = \langle edu.monash.jdbc.driver.mysql.2_1 \rangle$$
$$f = \langle \text{"jdbc driver for mysql",monash.mysql11.jar} \rangle$$
$$env = \langle JRE1.4, JRE1.5 \rangle$$
$$type = \langle passive \rangle$$
$$meta = \langle author: wsmith, version: 2.1, license: GPLv2, dependencies: \phi \rangle$$

Figure 3-5: Example capability (for JDBC driver)

A capability as described above is an *atomic* capability. It is possible to combine several capabilities and construct a *compound* capability. A compound capability with k atomic capabilities is therefore denoted as $c_k^+ = \{c_1, c_2, \ldots, c_k\}$ where $c_i \neq c_{i+1}$ and $k \in \mathbb{Z}^+$.

Capability Query

When an agent itinerary is constructed, it is necessary to be able to specify required functionality without referring to or being aware of a particular capability instance which implements the needed functionality. This is also relevant as a given VERSAG *function* can be implemented by multiple capabilities. We define the

concept of a *capability query*, encapsulating a *function description* and *execution environment* for this purpose.

Definition 4:

A capability query cq is represented as a pair $\left(F_{cq}^{desc}, E_{cq}\right)$ where, $F_{cq}^{desc} \subset \boldsymbol{\mathcal{F}}^{desc}$ and $E_{cq} \subset E$. Here, $\boldsymbol{\mathcal{F}}^{desc}$ denotes the set of all function descriptions (from Definition 2), and $E = \{e_1, e_2, \ldots, e_p\}$ denotes all possible execution environments (as in Definition 3).

In our example scenario, Bob's agent requires a database driver capability in order to connect to a MySQL database server. The agent is also aware that it needs this capability at a device which has Java Runtime Environment version 1.4. Figure 3-6 shows a capability query encapsulating this information.

$$f^{desc} = \langle \text{"jdbc driver for mysql"} \rangle$$
$$env = \langle JRE1.4 \rangle$$

Figure 3-6: Example capability query (for JDBC driver)

Match

Given an agent itinerary referring to capability queries, it is necessary to find capability instances that match these queries in order to execute the itinerary. We define a function *match* for this purpose.

Definition 5:

Given a capability query $cq = \left(F_{cq}^{desc}, E_{cq}\right)$ as in Definition 4, and a capability $c = (u_i, F_c, E_c, t_i, M)$ as in Definition 3, a function $f_{match}(c, cq)$ returns a boolean status of $true$ if capability c matches capability query cq or $false$ otherwise.

The matching algorithm is shown in Figure 3-7. We define an operator "$\oplus$" to compare between two given function descriptions or execution environments. In this research, since we use string/keyword descriptions, the "$\oplus$" operator for comparison is equivalent to the "equals" ("=") sign.

For a capability to match a capability query, the query's full set of *function descriptions* and *environments* must be supported by the capability. In our example, the JDBC driver capability shown in Figure 3-5 matches the capability query of Figure 3-6 as it provides the required functionality and also supports the required execution environment.

Algorithm: f_{match}

Input: Capability $c = (u_i, F_c, E_c, t_i, M)$, capability query $cq = \left(F_{cq}^{desc}, E_{cq}\right)$

Output: Boolean

1 **For each** f_i^{desc} in F_{cq}^{desc}
  For each f_j^{desc} in F_c^{desc}
    If $f_i^{desc} \oplus f_j^{desc}$
      Terminate inner loop //match found for f_i^{desc}
    End if
  End for
  Return $false$ //no match for f_i^{desc}
End for

2 **For each** e_i in E_{cq}
  For each e_j in E_c
    If $e_i \oplus e_j$
      Terminate inner loop //match found for e_i
    End if
  End for
  Return $false$ //no match for e_i
End for

3 Return $true$

Figure 3-7: VERSAG match algorithm

Activity

The concept of an activity represents the unit of work an agent has to carry out at a particular location. In order to carry out an activity an agent needs to execute one or more capability operations (O^c) in sequential order. An activity excludes *move* and *terminate* operations which are performed on the agent rather than on a capability.

Definition 6:

An activity a with k operations is denoted $o_j(c_i) \cdot o_{j+1}(c_{i+1}) \cdot \ldots \cdot o_{j+k-1}(c_{i+k-1})$ where the "·" operator indicates that operations are to be executed sequentially

from left to right. Here, $c_i \in C$ the set of all capabilities, and $o_j \in O^c$ the set of capability operations from Definition 1, and $i, j, k \in \mathbb{Z}^+$.

While we have defined an activity with the aid of capabilities, it is possible to specify activities using capability queries as well. In our example scenario, the activity that Bob's agent has to carry out to search a database can be represented with the aid of capability queries as follows:

$$a_{DBsearch} = get(cq_{dbsearch}) \cdot start(cq_{dbsearch}) \cdot discard(cq_{dbsearch})$$

The activity specifies that the agent should first acquire a *dbSearch* capability, start it and then discard it at the end.

Location

The concept of a location is central to modelling mobile agents [PKA07]. We define it as a place that an agent can visit and contains the necessary runtime environment for the agent to exist. A location has an associated *computing device* which does not change and an associated *physical location* which may change over time (i.e. due to physical movement of the device). In our definition of a location, we do not model these attributes. However, it can be expanded to include these and to support the definition by Price et al. [PKA07].

Definition 7:

A location l_i represents a place to which a VERSAG agent can migrate and execute at. The set $L = \{l_1, l_2, \ldots, l_q\}$ where $q \in \mathbb{Z}^+$ denotes all available locations.

Cost Element

Different types of costs incurred during itinerary execution are represented using cost elements. A cost element is typically (but not always) related to the use of computational resources. The "cost" associated with a cost element is measured in integer values. This is a design decision of VERSAG which has the consequence that any non-integer cost values have to be converted to an integer representation before use.

Definition 8:

A cost element q_i is denoted as a pair $\left(q_i^{name}, q_i^{value}\right)$ where q_i^{name} represents a unique name and $q_i^{value} \in \mathbb{Z}$ represents a cost value which can be associated with the cost element. We represent the set of all cost elements as $Q = \{q_1, q_2, \ldots, q_r\}$ where $r \in \mathbb{Z}^+$.

Possible cost elements include network load, time, CPU cycles, memory requirements, permitted number of executions and result accuracy. A measured network load of 1.3MB can for example be represented as $\langle \text{network load}, 1331 \rangle$ while the cost element without any value associated would be $\langle \text{network load} \rangle$.

In chapter 4, we describe how VERSAG agents are faced with various decision making steps during itinerary execution and propose a cost model to enable cost-efficient decisions in the face of multiple criteria. The concepts of cost elements and constraints defined here are central to building this cost model.

Relative Constraint

An agent's itinerary execution incurs costs in terms of different cost elements. If the agent is aware of the relative importance of different cost elements, it can attempt to minimize the more critical cost elements. A relative constraint is defined for this purpose.

Definition 9:

Let $q_i \in Q$ be a cost element as specified in definition 8. The relative constraint con^R is then denoted by a set $\{w_1, w_2, \ldots, w_r\}$ where $1 \leq i \leq r$ and $w_i \in \mathbb{N}$ represents the weight allocated to cost element q_i.

A relative constraint is applicable to an itinerary or an activity. In our example, when Bob's agent is searching the Intranet with applicable cost elements $\{time, network\ load, precision, CPU\}$, their relative weights could be represented as $con^R = \{2, 1, 1, 0\}$. This indicates that the cost element $time$ is twice as important as $network\ load$ and that $precision$ is as important as $network\ load$ while the cost element CPU (relative to the other factors) is of no importance.

Absolute Constraint

An absolute constraint places limits on permissible values for a cost element.

Definition 10:

Let $q_i \in Q$ be a cost element as specified in definition 8, $V_i \subset \mathbb{Z}$ be a set of permissible values for cost element q_i and $owner_i$ represent a location, an activity or an itinerary. Then, an absolute constraint con^A which is applicable to $owner_i$ is denoted by a triple $(owner_i, q_i, V_i)$. We denote the set of s absolute constraints as $R = \{con_1^A, con_2^A, \ldots, con_s^A\}$. Here $s \in \mathbb{Z}^+$.

An absolute constraint is applicable to an itinerary, an activity or a location. If, in our example, Bob requires his agent to complete the assigned tasks within 5 minutes (i.e. 300 seconds), an absolute constraint can be applied to the agent itinerary I as follows: $con_1^A = (I,\ time, [0..300])$. Furthermore, if Bob's smart phone can only assign a maximum of 16MB memory to running agent capabilities, this could be specified as follows: $con_2^A = \left(l_{smartphone}, memory, [0..16m]\right)$.

Itinerary

A VERSAG agent's primary task, as previously mentioned, is to execute an itinerary assigned to it. An itinerary specifies a list of places the agent has to traverse and activities to execute at each location [PKA07]. A VERSAG agent itinerary is similar to that described in the ITAG language [LZY01] with the main exception that a VERSAG itinerary belongs to a single agent. Furthermore, a VERSAG itinerary is a sequential structure. Parallel and non-deterministic actions cannot be represented in an itinerary and have to be incorporated inside capabilities.

Definition 11:

Let l_i denote a location (as in Definition 7), a_i denote an activity (as in Definition 6), con_i^R denote a relative constraint (as in Definition 9) and R_i denote a set of absolute constraints (as in Definition 10). Then, the 4-tuple $\left(l_i, a_{l_i}, con_{l_i}^R, R_{l_i}\right)$, which we term an itinerary step, represents execution of activity a_{l_i} at location l_i

with relative constraint $con_{l_i}^R$ and limited by absolute constraints R_{l_i}. An itinerary I is defined as follows:

$$I = \left(l_1, a_{l_1}, con_{l_1}^R, R_{l_1}\right) \cdot \left(l_2, a_{l_2}, con_{l_2}^R, R_{l_2}\right) \cdot \ldots \cdot \left(l_n, a_{l_n}, con_{l_n}^R, R_{l_n}\right)$$

Here, $n \in \mathbb{Z}^+$ and the "·" operator indicates that tuples are to be executed sequentially from left to right.

We use a tabular structure to illustrate an itinerary. A row in this tabular structure represents a 4-tuple $\left(l_i, a_{l_i}, con_{l_i}^R, R_{l_i}\right)$ with a column each for *location*, *activity* and the third column used to represent any applicable *constraints*. The third column can be omitted for simplicity when there are no constraints to apply. The relative constraint is usually specified per itinerary and is not represented inside the table.

Table 3-1 shows a section of the itinerary assigned to Bob's agent, in which it is required to upload his memo to the office Virtual Noticeboard (i.e. the steps illustrated in Figure 3-4). It is assumed that other capabilities which provide the agent with context-awareness, adaptation reasoning ability and capability exchange ability are already running. According to the itinerary, the agent first discards the *Editor* and *GUI* capabilities which are no longer required and then migrates on to PC-1. Once on PC-1, it searches for a client capability to access the virtual noticeboard. Once retrieved, the agent runs it to carry out the task. The memory constraint is generated by the context-awareness capability while the time constraint is user assigned. The time constraint will be taken into consideration during the *get* operation in selecting a suitable *VNClient* capability. Constraints and their effect on agent behaviour are further investigated in chapter 4.

Table 3-1: Bob's itinerary in tabular format

Location	**Activity/Operations**	**Constraints**
smartphone	discard *Editor*, discard GUI_1, move PC-1	(memory, [0..16m]) (time, [0..300])
PC-1	get *VNClient*, start *VNClient*, stop *VNClient*	(time, [0..300])

Application specific behaviour is displayed by an agent as a result of the itinerary it executes. It is possible for an agent's itinerary to be updated or changed before it is executed, during execution or after completion. Thus, an agent can be long-lived and take on different roles during its lifetime by running different itineraries over time. An agent resides in an idle state when it does not have an itinerary to carry out.

It is not expected that itineraries would be manually developed by end users. Users would instead issue high-level requests, possibly through user interfaces customized for a particular type of request. These requests would then be converted to itineraries by a component attached to the agent. It is also possible for this responsibility to be handed to an external entity, which generates an itinerary based on the user request and then sends the generated itinerary to the agent. Thus, it is possible to represent itineraries using a machine friendly syntax such as XML (Extensible Markup Language) or in a binary format when compact size is important. This thesis does not focus on the selection of an optimized format for itinerary specification.

VERSAG Adaptive Agent

A VERSAG agent has the ability to perform the primitive operations given in definition 1. The agent's primary task is to execute an itinerary assigned to it. To this end, it carries/acquires a set of capabilities which are required to fulfil the tasks specified in the itinerary. We define a VERSAG adaptive agent based on these three concepts.

Definition 12:

Let A be an adaptive agent. Then, A is represented as a triple (I^A, O^A, C^A) where,

I^A is the agent's itinerary as in definition 11,
O^A denotes the set of operations as in definition 1, and
C^A denotes the set of capabilities that the agent is carrying.

In this section we introduced the conceptual architecture and formally defined the key concepts of our proposed VERSAG agent framework. These concepts are used in subsequent descriptions of the framework, its algorithms and advanced features.

Next, we describe how the itinerant behaviour of an agent is implemented in the agent kernel.

3.4 Itinerary Execution with the Agent Kernel

A VERSAG agent at its most basic level is itinerary driven. This itinerant behaviour is implemented by the agent's kernel whose main responsibility is to retrieve itinerary commands from the itinerary service module and execute them. To achieve this objective, the kernel implements the "process coordinator" [Gor06] pattern as shown in Figure 3-8.

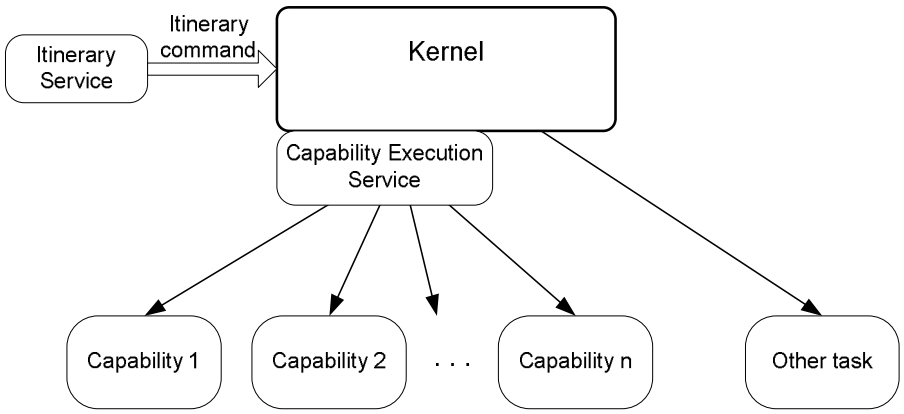


Figure 3-8: The kernel implements the process coordinator pattern

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. Figure 3-9 presents an algorithm describing such an execution cycle.

Each itinerary command has an accompanied *status* attribute which is updated as its processing proceeds. The principal supported status values are as follows:

- CREATED indicates that the itinerary command has been created but has not yet started execution.
- STARTED indicates that the command successfully started execution.
- EXECUTING indicates that command execution is in progress.
- EXECUTED, the command has completed execution.

- FAILED_X indicates a collection of states to represent miscellaneous failures where "X" is used as a wild card. One failure state FAILED_NO_REQUESTER_SERVICE indicates that a *get* operation failed because a running capability exchange service could not be found.

The algorithm steps can be further described as follows.

Step 1: The kernel starts by updating the previous command's status.

Step 2: Check that the current command is not empty and that the previous command has completed (or started in the case of a *cyclic* or *passive* capability). If not, the kernel blocks itself waiting for either completion of the previous command or an itinerary update.

Step 3: If the previous command failed, call a failure handling procedure to take appropriate action.

Step 4: If the current command contains a *move* or *terminate* operation, the kernel makes use of the interface provided by the base agent to carry it out. Before a move, the kernel saves details of currently running capabilities so that they can be restarted at the destination. A *discard* operation is easily implemented by removing the capability concerned from the agent's repository. However, it is necessary to check whether it is already running and if so, to stop it prior to discarding. *Start* and *stop* operations are carried out via the capability execution service. A *get* operation triggers the process of searching for, selecting, acquiring and saving a capability in the agent's repository. The kernel delegates carrying out this process to the agent's capability exchange service. Therefore, an instance of this service has to be already running for the agent to successfully execute a *get* operation. Once an operation is invoked, the command's *status* attribute is updated according to the result of the operation.

Step 5: Assign the *current* command to the *previous* attribute before proceeding to the next execution cycle.

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
```

Figure 3-9: VERSAG agent kernel execution algorithm

Having looked at how an agent is driven by an itinerary which supports six primitive operations, next we proceed to further examine capabilities which are the core distinguishing feature of VERSAG.

3.5 VERSAG Capabilities

A capability is a key feature of the VERSAG framework. They provide agents with different behaviours and are software components that can be shared and reused amongst agents. Since they can be shared amongst agents, capabilities also provide a fine-grained entity for migration and reuse in VERSAG based agent systems. 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.

In the remainder of this chapter starting from this section, we present a detailed examination of various aspects of VERSAG capabilities. We start off by identifying the requirements of a capability model and then describe the life cycle of a capability inside a VERSAG agent. Next, the peer capability sharing feature is described. The final two sub-sections look at issues related to discovering and describing capabilities.

3.5.1 Capability Model

A capability is essentially a reusable software component [CoH01]. A VERSAG agent gains application-specific behaviours and advanced features such as context-awareness and self-adaptability by executing capabilities. The *capability execution service* provides the runtime environment for capabilities to execute and is a key module of the agent. It allows running multiple capabilities in parallel, providing each with its own execution space, stopping them when required and also provides mechanisms for capabilities to communicate amongst themselves. Details on how capabilities interact with other software elements and the runtime environment required to support their execution are defined in a *capability model* (analogous to a component model as defined in [CoH01, WeS01]). Presented below are requirements and desirable features of a capability model for use within 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, allowing 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 would allow the capability to interact with other entities as needed.
- *Communication*. A capability should be able to communicate with other capabilities when needed and to 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 and 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 is dependent on version 0.9 of capability Z while Y is dependent on version 1.1). Evolution support is therefore a desirable feature of a capability model.
- *Standards*. A capability model underpinned by standards would 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 standards based approach is also desirable.

In our implementation (described in chapter 5), we make use of these requirements when selecting a capability model implementation for the VERSAG prototype.

3.5.2 Capability Life Cycle

We now define a set of life cycle states that a VERSAG capability will proceed through as it executes within an agent. It should be noted that the life cycle provided by a capability model implementation may have to be adapted to suit what is described herein. The first step in a capability's execution is for the capability execution service to load it from the agent's repository and create an executable instance of it.

- Upon initial creation, each capability is in a CREATED state.
- When the agent starts running a *cyclic* or *passive* capability, it moves to the STARTED state.
- When a *oneshot* capability starts running, it moves to the EXECUTING state.
- *Passive* and *cyclic* capabilities have to be explicitly stopped for them to move to the EXECUTED state. A *oneshot* capability moves to the EXECUTED state once its execution completes (without needing any external input).
- If starting a capability fails, it moves to a FAILED state. FAILED states are used to represent different causes of failures. It is also possible for a STARTED or EXECUTING capability to move to a FAILED state.

These state transitions are illustrated in Figure 3-10 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 until it is discarded. These status values correspond to the itinerary command states described in section 3.4.

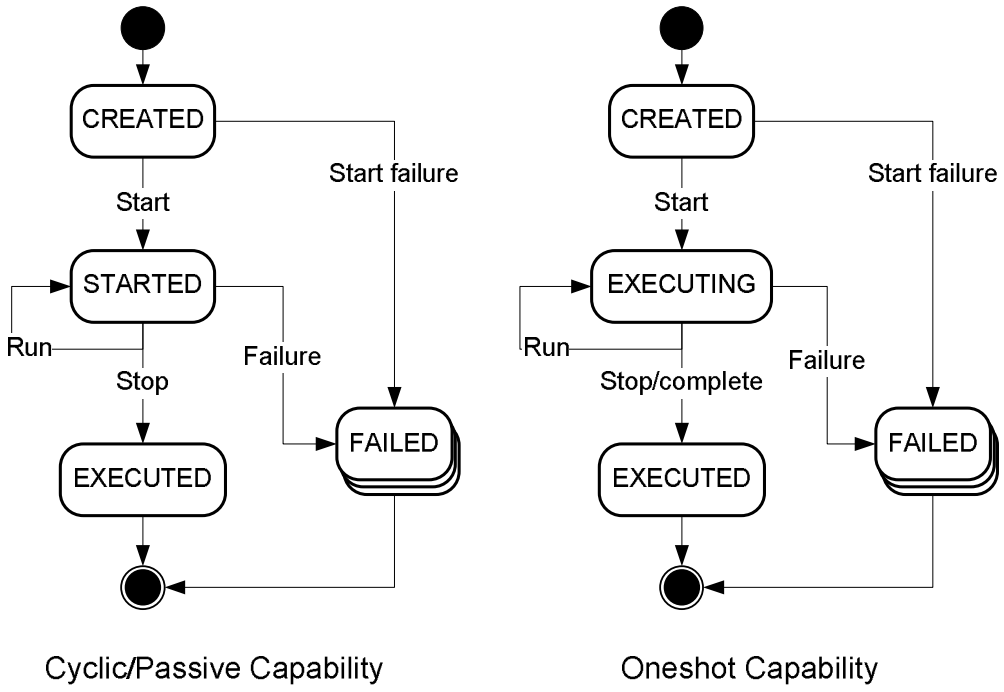


Figure 3-10: State transitions for capabilities

In the next section we leverage our component-based agent architecture with the ability to dynamically search for and acquire capabilities from peer agents.

3.5.3 Peer Capability Sharing

It was shown in Chapter 2 that in current systems compositionally adaptive mobile software agents often depend on a centralized component source. For example, in Dynamic Agents [CCD99], an agent downloads new programs (the equivalent of VERSAG capabilities) from a remote URL (Uniform Resource Locator), where the URL is specified together with the task request. Similar use of a remote component repository could be seen in PB3A [DPK00] and SCPA [FCL08].

Such dependence on a centralized component source is a disadvantage, especially in pervasive computing environments where dynamic variations in the environment can disrupt communication with the component source (e.g. due to network connectivity issues and possible source failure [NiL04, SSP04]). Consider, for example, a scenario where a group of agents form an ad-hoc network but are out of reach of the designated capability provider. 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. A compositionally adaptive agent depending on a single component source can therefore experience performance degradation or even 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, the disaster zone context exploring mobile agents described in chapter 1 run on severely resource limited nodes and cannot afford the luxury of frequent and large data transfers. The ability of agents to request and acquire capabilities from multiple sources including their peer agents becomes highly valuable in such situations.

GAMA [AmF05] proposes a decentralized approach in which each location has a component repository. While network disruptions do not affect component acquisition in this scenario, an agent is limited to only the components available in the repository at its current location.

VERSAG proposes a peer-to-peer variant of the decentralized approach where each agent is a potential capability supplier. Thus, agents are effectively able to share capabilities amongst each other. Peer capability exchange overcomes the above mentioned weaknesses of a centralized component source. Furthermore, acquiring capabilities from a nearby peer instead of a distant provider has the advantage of reducing network traffic by preferring local communication instead of global communication [GZK08a]. For example, in the hypothetical scenario of section 3.2, Bob's personal assistant agent first migrates to the office intranet and then asks nearby agents for necessary capabilities, thereby reducing the distance that capabilities have to be transferred. In this instance, all agents within the high-speed LAN of Bob's office are considered as "nearby agents".

While it overcomes most of the issues associated with the centralized approach, we recognize that peer capability sharing introduces new concerns. The distributed and decentralized nature of peer capability sharing increases 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 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 [MKL02, StW05, TsR03] and have been addressed in a multitude of ways. While addressing these issues is not the focus of the current research, in sub-section 3.5.4 we present a brief discussion on capability discovery as it pertains to VERSAG. Further, in [GLZ09] we propose the use of agent teams rather than purely ad hoc sharing of capabilities in order to improve the performance of capability sharing agents as an extension of the VERSAG approach. 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.

A significant benefit brought about as a result of peer capability sharing is efficient migration of agents, which we describe next.

3.5.3.1 Efficient Migration in VERSAG

Here we describe how VERSAG's capability sharing can be used to improve agent migration efficiency of systems built on top of existing mobile agent toolkits with fixed migration strategies. Capability based adaptation of agents introduces a new layer of code mobility to VERSAG agents, and it is possible to implement efficient migration mechanisms as described in chapter 2 at this level. Since application-specific functionality are implemented as capabilities, and constitute the bulk of the code to be transferred, such improvements can be considerable even though the base agent's code migrates happens as per the rules of the underlying agent toolkit. Mechanisms in VERSAG which lead to efficient migration (similar to those seen in the Kalong mobility model described in chapter 2) are as follows:

- An agent decides when to acquire, retain and discard capabilities needed to fulfil its itinerary. This effectively leads to adaptive transmission of code (in the form of capabilities).

- The ability of agents to store and serve capabilities to other agents is similar to having multiple code servers, and can also be viewed as a form of code caching.

Let us next illustrate this situation using our application scenario of Bob's personal assistant agent. Figure 3-11 illustrates Bob's personal assistant agent migrating to a new agency and using capability exchange to improve migration efficiency.

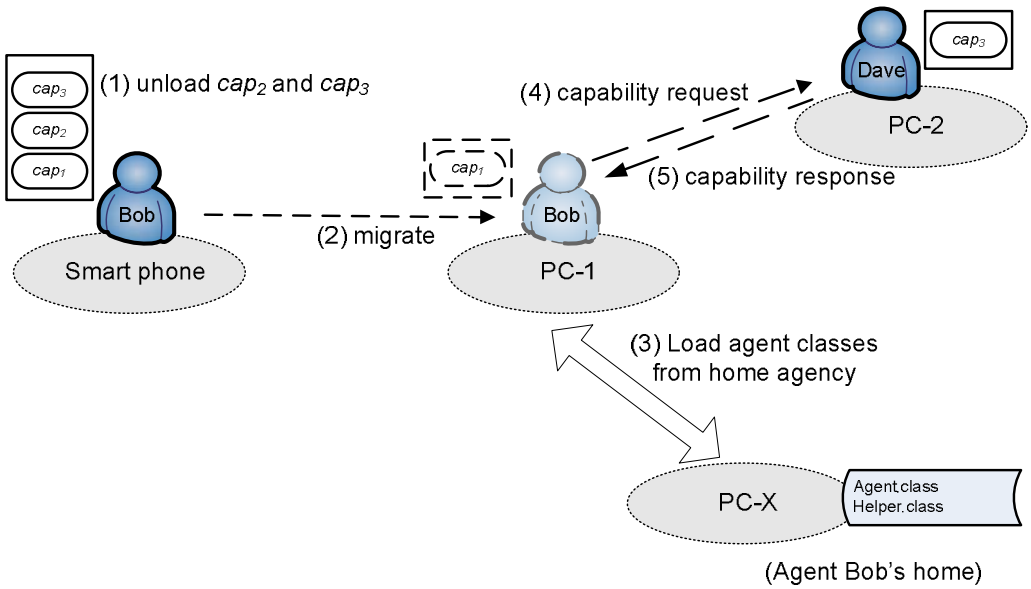


Figure 3-11: VERSAG agent migration on a platform with a pull migration strategy

Agent Bob has to migrate from the *smartphone* to *PC-1*. Bob has 3 capabilities cap_1, cap_2 and cap_3, of which cap_2 is already used and no longer needed. Agent Dave, residing at an agency close to Bob's destination *PC-1*, has cap_3. Therefore Bob first unloads cap_2 and cap_3 and migrates carrying only cap_1 with it. The base classes of Bob, needed to instantiate the agent at the destination, should be acquired by the agency based on the rules of the agent toolkit. A pull strategy is assumed and code is requested from Bob's home agency (i.e. *PC-X*). Since all VERSAG agents are instantiated from the same code (i.e. homogeneous from a program perspective), it is the same set of code that is required for all agents, and may be already available if another VERSAG agent has been to the same agency and the toolkit uses code caching. Finally, Bob requests cap_3 from agent Dave and acquires it. Steps 1, 4 and

5 shown in the diagram are carried out by VERSAG where as steps 2 and 3 happen according to the rules of the agent toolkit.

In this scenario, a standard mobile agent with the functionality represented by the three capabilities coded into it would have had to migrate carrying everything. If the underlying toolkit uses a *push-all* (or *pull-all*) strategy, the code for all three capabilities would be transmitted, even though cap_2 is no longer needed and cap_3 is available nearby. In case of a *pull-as-required* strategy, still cap_1 and cap_3 code would be transmitted unit by unit. With its capability sharing, VERSAG agents as shown above can select the most efficient approach to transmit code and achieve efficient code migration on standard mobile agent toolkits which do not provide such features. In Chapter 6, we present empirical evaluations which verify the performance benefits achievable through this approach.

Peer capability sharing of VERSAG necessitates that agents be able to discover and reason about capabilities. This in turn entails that capabilities should be described in a manner which allows automated reasoning on them. Thus, two significant concerns associated with peer capability sharing are discovering and describing of capabilities. There already exists a large body of work in this space and VERSAG leverages this body of work rather than propose new mechanisms of its own. In the next two sub-sections we present an overview of the issues involved and discuss what choices are available within the VERSAG context.

3.5.4 Discovering Capabilities

Service discovery is a significant research challenge in pervasive computing and has been the focus of much investigation in both academia and industry over the years [KiF02, ZMN05]. A service discovery system, according to Kindberg and Fox, [KiF02] is able to dynamically find services that meet the requirements of an entity that enters a pervasive environment. Correspondingly, a VERSAG agent needs to find capabilities (i.e. software components) that meet its requirements from the environment that it is located in. Due to similarities in the two situations, VERSAG agents are able to make use of many of the developments in service discovery for its capability discovery. Similarly, resource discovery has been thoroughly investigated

within the peer-to-peer computing paradigm [MKL02, TsR03] and solutions developed therein are applicable to VERSAG as well.

However, most current agent toolkits provide agent lookup or directory services. For example, JADE [BCP03] provides a directory service implemented via an agent termed the Directory Facilitator, which complies with the FIPA Yellow Pages service [BCP03]. Therefore, VERSAG only provides infrastructure for querying, selecting and obtaining capabilities and relies on the underlying agent platform services for discovery of capability supplying agents. Thus, each VERSAG agent that is willing to accept and serve capability requests from other agents registers itself as a 'capability provider' with the platform directory service.

Once the potential capability providing agents have been identified, an agent needs to query them for the required capabilities, accept their responses, select a suitable provider and request the capability from that provider. Figure 3-12 presents a graphical interpretation of the sequence of steps Bob's personal assistant agent follows for this purpose. Bob first asks peer agents (agents Dave, Ann and Tim) whether they have capability instances matching capability query cs_x. Two peers respond saying they have compatible capabilities after which agent Bob deliberates on which of them to select and then requests the capability from the selected peer agent (agent Dave).

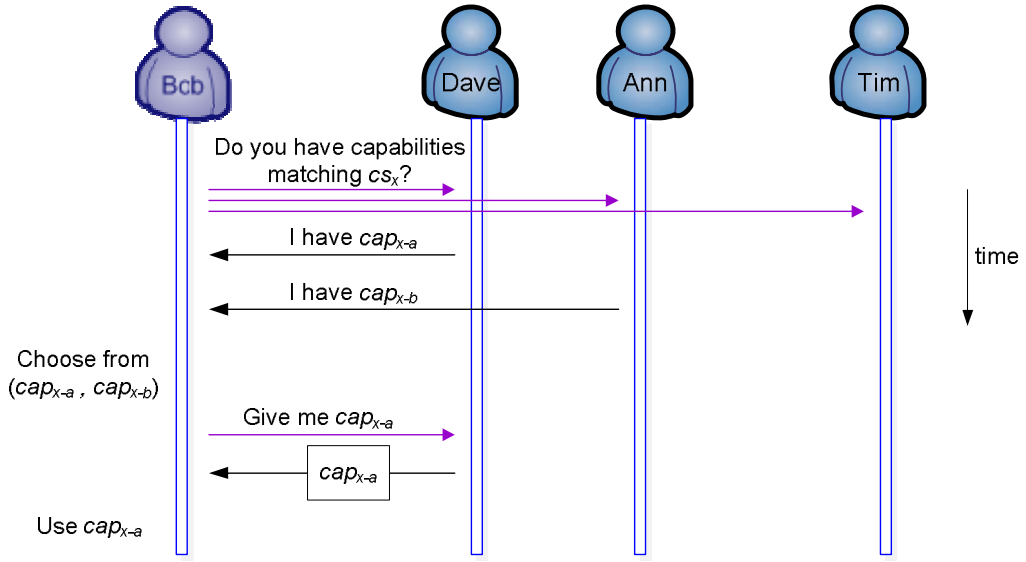


Figure 3-12: Capability search and acquisition without aid of a directory service

In the above scenario, the directory service was not used for locating capabilities. We note however that it is possible for agents to publish details of capabilities they contain with the directory service. Then, agent Bob can directly find out which agents contain matching capabilities from the directory service. While reducing the number of agents to query, it results in the provider agent having to frequently update the directory service with details of capabilities it contains. It is possible for a VERSAG agent to switch between these two approaches or to change its capability exchange protocol altogether by changing its capability exchange module.

A key related issue in discovering and sharing capabilities is describing capabilities in a manner that allows agents to reason about them and make suitable selections. The next sub-section is devoted to examining the issues related to describing capabilities.

3.5.5 Describing Capabilities

VERSAG capabilities need to be described in a manner that allows automated reasoning to be carried out on them. Four instances where this is necessary are as follows.

- *Search for matching capabilities*: During itinerary execution, an agent searches for capability instances that match its capability requirements. An agent that receives a search request has to compare functions and environments in the query with the capability instances it has and then reply with details of matching capabilities.
- *Select best capability instances*: The capability requesting agent, once it has received details of matching capability instances from its neighbours, has to sort these according to desirability and then acquire the most desirable set of capabilities for its purpose. This selection makes use of meta-data associated with the capability instances to check how they meet applicable constraints and estimates various costs.
- *Compose compound capabilities:* In some situations, it is not possible to find a single capability which can fulfil a required function. In such situations, it is sometimes still possible to achieve the required function by using several

capabilities together, thus forming a compound capability. In order to do this, it is necessary for an agent to be able to reason about the functions provided by each of the capability instances concerned.

- *Itinerary generation*[4]: Given a high-level user request, the agent should be able to build a detailed itinerary which can fulfil it. This requires breaking down high level tasks into smaller functions and allocating them to capability queries that the agent should acquire and execute.

Thus, agent capabilities should be described to support automated reasoning and matching required at various stages. In previous research on compositionally adaptive mobile software agents (e.g. Dynamic agents [CCD99], PB3A [DPK00], GAMA [AmF05]), component (i.e. capability) description and matching has not been given prominence, and defining components in a manner suited for reuse in a wider scope was not emphasized.

A VERSAG capability, as mentioned earlier, is a reusable software component. According to Pressman [page 736 of Pre01], an ideal description of a reusable software component should encompass its *concept*, *content* and *context* as defined in the 3C model [Edw92, Tra90]. Here *concept* is "*a description of what the component does*" [Whi95], including the interface specifications as well as its semantics. *Content* describes implementation details, which are not needed by mere users of the component, while *context* identifies the application domain. Canal et al. [CMP06] further state that while existing interface specification techniques are capable of ensuring syntactic matches, they are unable to ensure compatibility at behavioural and semantic levels. This therefore is an open research issue in component-based software development.

Similarly, describing services/software resources to allow automated discovery, reasoning and matching is accepted as a major research challenge [PTD08, SWK02] in fields such as autonomic computing [KeC03], grid computing [BFH03] and service-oriented computing [PaG03], where automated exchange of software resources is desirable. Thus, considerable research efforts have been exerted in these

[4] Itinerary generation and the use of compound capabilities are outside the scope of this thesis.

fields to describe resources in manners which enable their automatic discovery and selection. We briefly review some of these works below.

- *LARKS:* The Language for Advertisement and Request for Knowledge Sharing (LARKS) [SWK02] is one of the earliest and highly cited works on agent service description. It is an agent capability description language which allows heterogeneous agents on the Internet to advertise, request and match capabilities of agents. Dynamic matchmaking between agents is allowed using both syntactic and semantic information. (A capability in LARKS refers to a service provided by an agent and not to a software component as in VERSAG.) A LARKS specification is a *frame* with different slots describing the context, data types, input and output, constraints, semantic descriptions and a textual description. Semantic descriptions use domain ontologies written in their concept language ITL (Information Terminological Language) [SLK98]. In addition to *exact* matches, LARKS provides two other types of matches: *plug-in* and *relaxed* which have fewer restrictions. The three types of matches are implemented using a combination of five filters. LARKS aims to provide a balance between the quality of matches and performance requirements of agents through its flexible approach.
- *OWL-S:* (OWL for Services [MBM07]). Service-Oriented Computing [PaG03] sees services as the building blocks of larger applications, and web services are the manifestation of these concepts in the Internet [BHM04]. However, automated composition of applications from services has been hampered by the lack of semantic descriptions in the standard Web Service Description Language (WSDL) [GLX05]. WSDL specifications limit themselves to describing the syntax of services and how to access them, leading to the necessity of further descriptions and human intervention for service composition. OWL-S [MBM07] is an ontology developed using the Web Ontology Language (OWL) to fill this gap of semantically describing web services. OWL-S consists of three main sub-ontologies:
    - A *service profile* describing ‘what the service does’
    - A *process model* describing ‘how the service is used’ and
    - A g*rounding* describing ‘how to interact with the service’.

A service profile semantically describes the type of service, functions provided by the service (corresponds to VERSAG functionality) and non-functional aspects which include areas such as security, accuracy, timeliness and costs. The service grounding provides a mapping between OWL-S and WSDL, paving the way for web service access by a software client. Similar to LARKS, OWL-S can be used for dynamic matchmaking between service requesters and providers. For example, Guo et al. [GLX05] describe web service capabilities using OWL-S service profiles and use a matching algorithm with five levels of matches: *exact*, *plug-in, subsume, intersection* and *not relevant*.

- *WSMO:* The aim of the Web Service Modeling Ontology (WSMO) [RKL05] is to semantically describe web services to enable their automated discovery, composition and invocation. For this purpose, WSMO defines a conceptual model and a formal language. It further defines four top-level elements as its key concepts.
    - *Ontologies* are used to provide semantic descriptions of elements in use.
    - *Web services* represent the providers of useful services and are semantically described using ontologies.
    - *Goals* are used to represent user requirements.
    - *Mediators* have the responsibility of solving interoperability issues between service providers and service requesters.

  The functionality provided by web services is expressible in WSMO and can be used for discovery and selection of services.

The above approaches are used for semantic description and reasoning about services that are to be accessed remotely. They could be adapted for reasoning about software components such as VERSAG capabilities with the addition of attributes to represent runtime features such as component size, execution constraints, software requirements, and hardware requirements (e.g. input/output devices).

While ontology based descriptions ease automation of reasoning, they tend to be computationally demanding [PrB08b, SKG09], and conflict with VERSAG's requirements of being lightweight to support resource-constrained environments. Also, peer agents that carry out the task of matching capability instances against capability specifications are doing it as a secondary task in addition to its main application-specific workload. Thus, these agents can only allocate a small amount of their resources for the matching task. Some approaches that could be used to address this conflict are as follows:

- Where possible, reasoning workload could be offloaded from agents to entities that have more resources. For example, an agent in a resource-constrained environment may ask another trusted agent that is idle at present to do the reasoning on its behalf.
- Agents could maintain records of previous reasoning histories and reuse them. Similar to capabilities, such information could also be shared among agents.
- Switch to simpler reasoning and matching mechanisms such as keyword matching whenever possible.
- Previous research attempts to support semantic awareness in resource-constrained environments can be incorporated into VERSAG. For example, Preuveneers and Berbers [PrB08b] propose an ontology encoding scheme that is compact and suitable for reasoning on such devices while the mTableaux algorithm by Steller et al. [SKG09] optimizes reasoning on devices with limited resources.

VERSAG agents can dynamically switch between these approaches by implementing different reasoning mechanisms as capabilities. For example, the "⊕" operator in the *match* functionality (from Definition 5) of an agent could be implemented as a simple String matching of keywords for use in resource-constrained situations and as an OWL-S based semantic matching with support for partial matches for use in other situations.

As stated earlier, developing new mechanisms for capability description and matching is outside the scope of this thesis., We envision that VERSAG will leverage any one of the works described herein depending on application and environmental requirements. In our implementation (described in chapter 5), we develop a version of capability matching for evaluation purposes that uses string/keyword based matching.

3.6 Summary

This chapter presented, as the main contribution of this research, our proposed VERSAG mobile agent framework which enables building dynamic pervasive applications and services. The proposed agents are compositionally adaptive and built on a lightweight framework which makes them suitable for use in heterogeneous environments with varying resource levels. VERSAG agents gain new behaviours at runtime through reusable software components termed *capabilities*. The two salient features of VERSAG agents are as follows.

- Six primitive operations have been proposed to model an agent.
- Agents acquire new capabilities from and share theirs with peer agents.

We then proposed a reference architecture for a VERSAG agent identifying the key modules that make up an agent. Next, the key concepts underpinning the framework were proposed and defined. This was followed by the development of the agent kernel module which included an algorithmic description of the agent's itinerant behaviour.

The later part of this chapter was dedicated to describing VERSAG capabilities and related issues. We started by identifying the requirements of a capability model and proceeded to describe the life cycle of a capability inside a VERSAG agent. Then, we described the peer capability sharing feature and how agent migration efficiency can be improved as a result of capability sharing. This was followed by an examination of issues related to discovering and describing capabilities. We identified that similar work in other fields such as peer-to-peer computing and service-oriented computing could be leveraged by VERSAG.

Thus, in this chapter we introduced the concept of VERSAG and proposed its core features which provide us with lightweight mobile agents that are highly versatile. The real strength of VERSAG in terms of its suitability for pervasive computing lies in the ability of agents to dynamically adapt to achieve greater autonomy based on application and environmental needs. Therefore, in the next chapter we present our proposed adaptation and decision making cost model for VERSAG agents. The cost model, which itself is a dynamically acquirable feature of an agent, enables agents to make cost-efficient adaptation decisions by taking into consideration multiple criteria that are either user/organization specified or brought about due to environmental constraints.

4 VERSAG Agent Adaptation and Cost Model

The previous chapter introduced and described our proposed Versatile Self-Adaptive Agent (VERSAG) framework. In this chapter we describe how VERSAG agents achieve greater autonomy through dynamic capability acquisition and how agents make cost-efficient adaptation decisions through the use of a multi-criteria cost model.

4.1 Introduction

VERSAG agents are compositionally adaptive and built on a lightweight framework which makes them suitable for use in heterogeneous/pervasive environments with varying resource levels. Reusable software components termed capabilities provide agents with application-specific functionality as well as behaviours which allow agents to increase their level of sophistication. A key innovative feature of our proposed framework is the ability for VERSAG agents to acquire and discard capabilities at runtime. The proposed framework has the following two salient features:

- *Simple primitive operations.* A VERSAG agent is an active mobile entity which can carry out six primitive operations. These primitive operations enable an agent to manage capabilities at runtime, acquiring, discarding and executing them based on needs.

- *Peer capability sharing*. Agents do not depend on a single capability source and can share capabilities they possess with peer agents.

A VERSAG agent can thus change its functioning/internal structure dynamically by acquiring and discarding capabilities appropriately, according to environmental and application needs. In chapter 3, we formally defined the key concepts underpinning VERSAG and described the feature of peer capability sharing. In this chapter we describe how VERSAG agents achieve greater autonomy through dynamic capability acquisition and how agents make cost-efficient adaptation decisions through the use of a multi-criteria cost model.

Mobility and intelligence[5] are desirable qualities in both pervasive computing applications and in software agents [PrB08a, TGM98]. However, they tend to be conflicting features as greater intelligence generally results in higher resource requirements and larger software components which in turn inhibit mobility [TGM98, Zas04]. The VERSAG agent framework aims to reduce this conflict by enabling agents to dynamically adapt their levels of sophistication, size and resource needs based on application and environmental requirements. To reach this goal, it is necessary for an agent to be lightweight and flexible in the first place. Thus, defining a VERSAG agent as a lightweight itinerant mobile entity with six primitive operations and the ability to dynamically attach/detach capabilities is an essential enabling feature. The primary adaptation actions currently supported in VERSAG are through its primitive operations of starting a new capability (*start*), stopping a currently executing capability (*stop*), discarding a capability (*discard*), acquiring a new capability (*get*), migrating to a different location (*move*) and terminating itself (*terminate*). Two other forms of adaptation an agent could undertake, but not considered in this thesis are changing the agent's base *platform specific agent* module implementation to support migration between heterogeneous agent platforms and changing the order of visits in the agent's itinerary. In this chapter, we describe how these simple itinerant agents can adapt to cope with varying and dynamic

[5] We use the term "intelligence" here in a broad sense to include features such as context-awareness and decision making cost models as well as classic intelligence features [Hay95].

environments and application needs by using a cost model to facilitate efficiency in this process.

While an agent's adaptation steps could be explicitly specified in its itinerary, to be successful when the environment conditions are dynamic and not known *a priori*, the agent should be able to autonomously decide on adaptation actions based on current situations. Towards this end, this chapter proposes and develops a cost model which helps an agent make cost-efficient adaptation decisions. The decision making criteria can be user/organization specified or brought about due to environmental constraints. Constraints which place limits on cost criteria (e.g. memory limit on a mobile device) are captured in VERSAG using *absolute constraints* (section 3.3.2) while the relative importance of different cost criteria to be considered are represented as a VERSAG *relative constraint* (section 3.3.2). In our approach, a key facet of this decision making revolves around how the cost model can be applied when an agent needs to select suitable capability instances from amongst many alternatives.

The rest of this chapter is organized as follows. Section 4.2 provides an overview of adaptation in VERSAG agents, describing how adaptation enables agents to deal with varying environmental conditions. Section 4.3 introduces the cost-efficient decision making feature and the multi-criteria cost based selection model. Key cost elements currently supported and techniques for estimating them are presented in section 4.4. Section 4.5 provides a short discussion on the extensibility of the cost model, and the chapter is concluded with a summary in section 4.6.

4.2 VERSAG Agent Adaptation

A basic itinerant agent in VERSAG is not aware of the application level semantics of the tasks it carries out or how well it is performing the allocated tasks, and does not have the ability to improve its behaviour as a result of such introspection. The agent instead needs to be given exact instructions (via its itinerary) on what actions to perform. Such basic itinerant agents are useful when the agent's sequence of actions can be decided in advance (e.g. carry out a pre-defined task in a stable environment) or when the agent is monitored and controlled by an external entity. Conversely, the ability to delegate tasks to an autonomous agent is a key benefit of the mobile agent

paradigm [BrR04], and is particularly useful when agents execute in dynamic heterogeneous environments, as encountered in mobile and pervasive computing. In this section we discuss how a VERSAG agent can adjust its mode of operation (ranging from a lightweight itinerant agent to a context-sensitive autonomous agent) and gain new features through acquiring appropriate capabilities depending on the context in which it is performing its tasks.

It was explained in the previous chapter that VERSAG agents gain their application specific behaviours by executing reusable software components termed capabilities. In addition to application specific functionality, capabilities can implement functionality which enhance the agent's sophistication. For example, the peer capability sharing feature of VESAG (i.e. *capability exchange service* in the agent reference architecture) is itself implemented as a capability. Thus, this feature can also be modified or removed from the agent as required. Similarly, a context-sensing capability can be attached to an agent to make the agent context-aware (i.e. *context service* in the agent reference architecture). Thus, the capability exchange mechanism of VERSAG can be leveraged to even enable an agent with advanced features such as learning and negotiation as long as they can be encapsulated as VERSAG capabilities.

Let us briefly consider how these adaptation features can be useful in an agent based application. In our example scenario, Bob's personal assistant agent when it resides on the smart phone, runs a *context-sensing* capability to monitor resource levels (e.g. battery level, network bandwidth, memory) on the device. If it observes that the battery level has decayed below a certain threshold, it requests the agent to migrate to another device (by altering the itinerary). Thus, with the use of this capability, the agent gains the ability to sense its environment and act accordingly. Furthermore, when Bob instructs the agent on the tasks he needs it to carry out, this is a high level request issued through a *user interface* (UI) capability. A separate capability generates a detailed itinerary from this high level request and allocates it to the agent. This *itinerary generator* capability (in collaboration with the *context-sensing* capability) determines the policy for handling capabilities for the current itinerary. Furthermore, it is the *itinerary generator* that decides the agent should discard

unwanted capabilities before migration over the wireless network. Once on the office network and searching for required capabilities, the agent sees that peer agents have multiple matching capabilities. At this point, an *adaptation decision making* capability comes into play and helps the agent make a cost-efficient selection.

We now describe two adaptation features of VERSAG agents mentioned in the above example: context-awareness and itinerary generation. It should be noted that these are illustrative only and that an agent may adapt its features and functions in any number of ways. Our focus on these two features is due to their importance for pervasive computing applications which is a primary motivation of VERSAG.

4.2.1 Context-Awareness as an Adaptation Feature

Context-awareness, as previously mentioned, is an essential feature in pervasive computing environments [Sat01, Zas04]. Thus, it is useful for mobile agents to be equipped with context-awareness when executing in dynamic environments. A VERSAG agent in such an environment can gain context-awareness by equipping itself with a suitable context sensing and awareness capabilities. We note that how the capability implements its sensing functionality varies according to environment and system requirements and is not central to the current discussion. The focus of our discussion is on how the capability interfaces and interacts with the rest of the agent's structure. If the capability is to continuously monitor the environment and directly influence the agent's behaviour according to contextual conditions, it needs to be implemented as a *cyclic* capability (which results in a separate execution thread within the agent) and should be able to modify the agent's itinerary and activities as necessary. As a result, the agent gains increased autonomy over its own itinerary specific behaviour through a more current and relevant awareness of its environment.

Alternatively, context sensing could be implemented as a *passive* capability which exposes its sensing functions for use by other capabilities. The component diagram of Figure 4-1 illustrates how such a passive context service can be used by other capabilities. In the diagram, the context service provides (i.e. implements) the *ContextServiceIF* interface through which other capabilities get access to its sensing functions. In the diagram, capabilities *Task A* and *Task B*, invoke the context service

through this interface. A capability which generates/updates the agent's itinerary based on environmental conditions is an example of such a context using capability. A key enabler for VERSAG agents to dynamically gain features which enhance its reasoning and autonomic abilities is its capability-centric architecture. The architecture not only lets capabilities to be dynamically acquired and executed within an agent but also allows them to communicate and interact with each other and with the core agent for increased effectiveness. Thus, this architecture can enable VERSAG agents to acquire or enhance their reasoning, autonomy, and intelligence level with dynamically acquired capabilities.

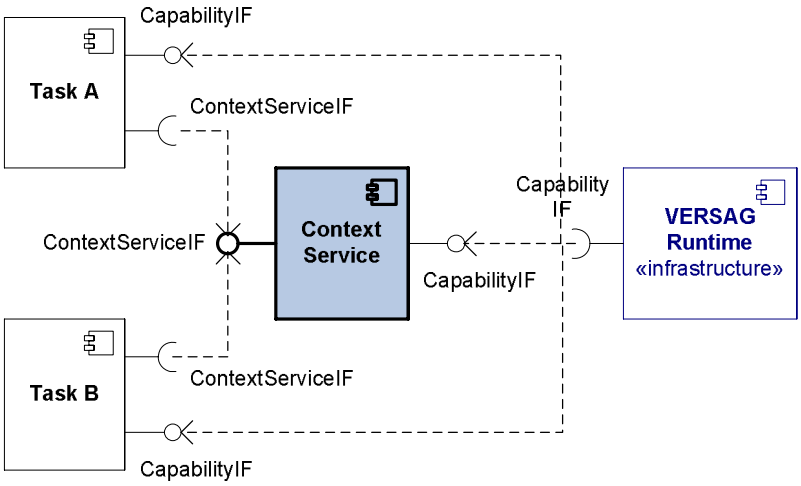


Figure 4-1: Component diagram of VERSAG context sensing capability

In the next sub-section we describe how itinerary generation could be incorporated into VERSAG agents in the form of capabilities.

4.2.2 Itinerary Generation as an Adaptation Feature

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 SQL [ISO08] select statements). 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. We assume that the generated itinerary:

- Specifies a sequence of *locations* for the agent to visit,
- Specifies the *activity* to be carried out at each *location* as a sequence of operations on *capability queries* (e.g. $stop(cq_1) \cdot start(cq_2) \cdot start(cq_3)$), and
- Defines a *Capability Management Policy* (CMP).

A *capability management policy (CMP) s*pecifies generic rules which enable a VERSAG agent to make decisions on capability management. It enhances the operation of an agent by indicating the points at which an agent should search for and acquire capabilities and when it should discard used capabilities. For example, a "light-travel" CMP recommends that capabilities are acquired just before they are used and discarded as soon as they are used. Table 4-1 illustrates an itinerary with this policy.

Table 4-1: A detailed itinerary based on the "light-travel" CMP

Location	Operations
loc1	**get *cap*$_1$**, start *cap*$_1$, stop *cap*$_1$, **discard *cap*$_1$**, move loc2
loc2	**get *cap*$_2$**, start *cap*$_2$, stop *cap*$_2$, terminate

Another simple CMP is "carry-all" which recommends acquiring all capabilities at the beginning of the itinerary and carrying them until the itinerary is completed. A slightly modified version of this policy is "carry-all-with-discard" where the agent discards a capability after its final use as illustrated in the itinerary of Table 4-2.

Table 4-2: A detailed itinerary based on the "carry-all-with-discard" CMP

Location	Operations
loc1	**get *cap*$_1$**, **get *cap*$_2$**, start cap_1, stop cap_1, move loc2
loc2	start cap_1, stop cap_1, **discard *cap*$_1$**, move loc3
loc3	start cap_2, stop cap_2, **discard *cap*$_2$**

Such static CMPs can be implemented as part of the itinerary generator capability. A CMP may also be dynamic, where the agent deliberates and updates the itinerary at runtime with *get* and *discard* operations. A dynamic policy makes use of information that becomes available as the agent executes (e.g. outcomes of previous results and context data) in decision making. Implementation of a dynamic CMP results in the itinerary being interspersed with *deliberate* operations as shown in Table 4-3. A *deliberate* operation refers to executing a *oneshot* type capability which is part of the itinerary generation process. At each such step, the agent's itinerary is examined and updated with acquire (*get*) and remove (*discard*) operations. A deliberation step may include the invocation of a context sensing capability (as seen in section 4.2.1) to obtain environmental information as a precursor to its deliberation.

Table 4-3: A detailed itinerary based on a dynamic CMP

Location	Operations
loc1	**start *deliberate***, start cap_1, stop cap_1, **start *deliberate***, move loc2
loc2	**start *deliberate***, start cap_1, stop cap_1, **start *deliberate***, move loc3
loc3	**start *deliberate***, start cap_2, stop cap_2, **start *deliberate***, terminate

Selection of a suitable capability management policy (CMP) is important to ensure that a user request is carried out in an efficient manner. For example, an agent which does the same task at multiple locations may opt to load the required capabilities once and carry them throughout rather than loading and discarding them at each location. In a situation where constraints arise due to limited network bandwidth, it

would be prudent to attempt as much as possible to find capabilities from nearby peer agents rather than carrying them around.

Itinerary generation clearly increases agent autonomy. It is however a complex process which involves capturing user input, decomposing the request into a sequence of primitive operations and identifying a suitable migration path for the agent. While user input can be captured with relative ease, for example via a user interface capability or through agent messages, other involved activities are more complex and their investigation is outside the scope of the current research. We note that existing work on mobile agent planning [BYK01, BKR99, DSW05, Moi98], planning [KyD00, NCL99] and itinerary generation [QiW01, YLY03] can be leveraged to implement itinerary generation in VERSAG.

While the itinerary generation process results in a detailed set of instructions for an agent to follow, an agent still encounters multiple decision making steps during itinerary execution. This therefore necessitates a VERSAG agent to be equipped with a cost-efficient decision making ability. The development of this cost-efficient decision making ability for VERSAG agents is a key contribution of this research and is the focus of this chapter. The following sections describe in detail how such decision making ability can be incorporated into VERSAG agents through the use of capabilities.

4.2.3 Summary

In this section, we presented two examples of how autonomous behaviour and the ability to cope with changing pervasive environments can be incorporated into a VERSAG agent through capabilities. These capabilities can be dynamically acquired by an agent as and when needed, and discarded when no longer needed. For example, when a VERSAG agent is required to migrate to a resource-constrained device as part of its execution, the agent can alter its itinerary to discard these capabilities before migrating to the resource-constrained device, and to re-acquire them after migrating back onto a device with more resources. Thus, for the duration that it is on the resource-constrained device, the agent converts itself into a lightweight itinerant agent.

It should be noted that the examples presented above are illustrative only and that an agent may adapt its features in any number of ways. For example, an agent can improve user interaction by acquiring a Graphical User Interface (GUI) capability or improve its ability to learn by acquiring a capability which implements a reinforcement learning [KLM96] strategy. Thus, a VERSAG agent can display intelligent, sophisticated behaviour in environments and domains where such features are required by using the appropriate capabilities and also be a simple and lightweight itinerant agent when such sophistication is not required. Central to this ability is the definition of an agent as a lightweight mobile entity with the ability to adapt at runtime by acquiring and discarding capabilities as necessary.

The remaining sections of this chapter focus on describing how decision making know-how can be incorporated into VERSAG agents as an adaptable feature which also leverages the notions of capabilities.

4.3 Cost-Efficient Decision Making

While a VERSAG agent's itinerary specifies a sequence of locations to visit, activities to carry out and a capability management policy to use, it still has to take many decision making steps during the course of its lifetime as a result of interactions with other agents, environmental changes (identified through the agent's context-awareness) and outcomes of activities carried out thus far. Such situations include the following:

- A dynamic capability management policy (CMP, see section 4.2.2) requires the agent to deliberate on capability acquisition/discarding during itinerary execution. Thus, each deliberation step is a decision making step.
- During capability acquisition, when several suitable alternatives are available in the environment, the agent has to make an appropriate selection based on some guiding criteria.
- The agent may have to change the CMP during itinerary execution in response to changing circumstances.

- The agent's migration path may have to be modified during itinerary execution as a result of changes in the environment and outcomes of previous steps.

At these decision making steps, it is desirable to be able to select a cost-efficient alternative when more than one acceptable alternative is available. We propose that such cost-efficient decision making know-how can also be represented as capabilities, thereby allowing these to be incorporated dynamically into the agent when needed.

A key decision making step identified above is capability acquisition (i.e. *get* operations). An agent searching for a capability to match a given query is likely to find more than one match from amongst its peers, each having different execution costs in terms of required time/memory/CPU due to the use of different internal algorithms, and different acquisition costs due to differences in network distance and involved monetary costs. In such a scenario, an agent should be able to compare and select the most cost-efficient capability instance. The following discussion on cost-efficient decision making for VERSAG agents is built around this use case. We later present in this chapter how our proposed solution/approach can be extended for use in other decision making situations.

The aim of the proposed cost-efficient capability acquisition feature is to let an agent select (from capabilities available in its neighbourhood) a set of capabilities that can fulfil its assigned activities while meeting other relevant constraints and minimizing relevant costs. The constraints and cost components applicable vary from one situation to the other and can be either explicitly specified or inferred based on environmental conditions.

This section is organised as follows. We start with a motivating example in the next sub-section and then introduce cost elements relevant to decision making. This is followed by a brief look at how cost-efficient decision making can be incorporated to a VERSAG agent as an adaptation feature. Then we present an algorithm that describes how the decision steps can be incorporated into an agent and develop a cost model for cost-efficient selection of capabilities. Subsequently, we describe methods

to estimate key cost elements and finally discuss how this model could be extended for use in other decision making situations detailed above.

4.3.1 Motivating Example

It was suggested earlier in this chapter that Bob's personal assistant agent uses an *adaptation decision making* capability to make cost-efficient selections when presented with multiple capabilities that can fulfil its needs. We present a detailed description of this scenario next and use it in the following sub-sections to illustrate the proposed solution.

Bob's agent is assigned to search for resources in the office Intranet for references to their m-commerce product. One sub-task identified within this is to search a MySQL database, and is represented as the activity $a_{DBSearch}$ to be carried out at location *DBClient-PC*. The activity is defined using three capability queries as shown.

$$a_{DBSearch} = start(\boldsymbol{cq_{dbdriver}}) \cdot start(\boldsymbol{cq_{dbclient}}) \cdot start(\boldsymbol{cq_{search}})$$

The query cq_{search} represents a task specific capability which searches a data source for some given content. It requires client capabilities which can talk to different types of data sources. A database client capability is represented by the capability query $cq_{dbclient}$. This in turn requires a $cq_{dbdriver}$ capability which enables it to interact with a specific database server type (e.g. MySQL). Consider the case where the agent searches its nearby peer agents for these capability queries and locates the following capabilities shown in Table 4-4.

Table 4-4: Sample capability sources for Bob's agent

Capability Query	Capability	Peer agent	Location
$cq_{dbdriver}$	$c_{gnu-mysql}$ $c_{larry-mysql}$ $c_{gnu-mysql}$	Dave Admin OSS	Dave-PC Support-PC Sourceforge
$cq_{dbclient}$	$c_{toadsql}$ $c_{frogsql}$	Dave Ann	Dave-PC Reception-PC
cq_{search}	$c_{searchV1}$	Bob	DBClient-PC

Since there is only one match for cq_{search} (i.e. $c_{searchV1}$) the agent must use it. However, for $cq_{dbclient}$ and $cq_{dbdriver}$ there are multiple matches and the agent needs to select one of each. Our aim with the cost-efficient capability acquisition feature is to help the agent select capabilities which minimize costs and meet any absolute/fixed constraints. In order to minimize costs, it is important to understand the various costs associated with particular capabilities. Thus, in the next section we describe our cost criteria.

4.3.2 Cost Criteria of Capability Acquisition

There are a number of cost criteria that are relevant for capability acquisition. For example, a fundamental cost element is the time required for completion of an itinerary/activity. We also consider the level of trust placed on a capability to be a cost element. Other cost criteria are imposed by the hardware environment (e.g. memory/CPU limits) and users (e.g. "minimize time consumption", "60% accuracy of results is sufficient"). The following are the important cost criteria that underpin our cost model:

i. *Time:* Time is a key quality metric in many mobile/distributed applications. The need to improve application throughput and response time, organizational policies, monetary concerns (e.g. to avoid entering a higher cost band when billed based on time) and real-time requirements of certain applications such as health monitoring and vehicular control [Pay86]

necessitate the need to maintain minimal time. Therefore, itinerary execution time is a crucial cost element that an agent needs to consider in its adaptation decision making. A VERSAG agent can use estimates of time consumption of different alternatives (i.e. capabilities) in order to make an informed decision.

ii. *Network Load:* VERSAG, as an enabling framework for mobile agents in pervasive computing environments, can encounter networks varying from high-speed local area networks (speeds in Gigabit ranges) to unstable ad hoc wireless networks with low speeds (in Megabit and kilobit range). Thus, network traffic generated when executing an itinerary is an important element to be managed. In general, excessive network traffic can lead to network congestion, performance degradation, and application failure. Similar to time consumption, network usage can also have direct monetary costs in some cases (e.g. access charges of Internet Service Providers). While a lower network load is desirable in general for better performance, it has less importance when the agent is executing in a high-speed network. Therefore, network load is identified as another key cost element that an agent needs to consider in its adaptation decision making.

iii. *Computational Resources:* A mobile agent encounters heterogeneous devices with different enabling opportunities as it migrates. Therefore, it is difficult to make design-time assumptions about the computational resources that would be available during its lifetime. For example, when Bob's personal assistant agent resides on his smart phone it has limited memory and processing capacity to work with. Available resources are further limited by having to share them with a number of other agents and processes. Thus, it is desirable for the agent to be able to adapt itself according to computational resources available at a given time and location. A VERSAG agent is able to achieve this goal by dynamically changing its constituent components (i.e. capabilities).

Capabilities with lower resource requirements are likely to lead to performance losses in other criteria such as time. Therefore, trade-offs amongst these conflicting costs need to be managed in decision making.

iv. *Accuracy of Output:* Accuracy of a software application is an indication that its output precision is adequate to meet its intended use [BBL76]. For example, the temperature sensing precision of a home weather station is not adequate to meet the sensing needs of an industrial chemical reactor. Situation-aware systems are able to dynamically change the precision and frequency of their functioning in order to conserve resources when required. In [DZK09], a smart phone wirelessly reads and analyses data from a body worn bio sensor and is able to reduce its sampling frequency in order to conserve resources (e.g. battery power) when the wearer's vital signs are normal.

The cost element of accuracy of output is included as a criterion for consideration in agent adaptation to reflect this requirement. This lets an agent to select a less accurate capability in order to improve on other more important cost criteria. We use the terms accuracy and precision interchangeably.

v. *Energy:* Energy is a scarce resource on battery powered mobile computing devices, and it is important that applications on such devices take measures to minimise energy consumption [FoZ94, Sat01]. Therefore, when a VERSAG agent's itinerary involves visiting such devices, energy requirements of capabilities as well as current available battery levels of the devices become important cost criteria for consideration in adaptation.

vi. *Policy requirements:* Agents which belong to a corporate entity have to follow organizational policies which, for example may require capabilities to be from a particular vendor, or meet certain standards. Similarly, if an agent belongs to an individual (such as Bob's personal assistant agent), there can be associated user preferences which place constraints on adaptation. While these are often strict rules to be adhered to, they may sometimes reflect flexible features [TeM01] than can be overlooked in order to meet other high priority cost criteria.

vii. *Trust:* Trust [GrS00] is an important component of distributed applications. In VERSAG, an agent needs to trust the remote agents that it interacts with

and the capabilities that it acquires for use. Therefore, trust is also an important cost criterion to be considered in capability selection.

viii. *Software Quality:* A VERSAG capability is a software program, and *software quality* attributes that apply to software products are applicable to capabilities as well. These in turn are of concern to the agent when selecting a capability. Thus, quality features such as usability, robustness, efficiency, and reliability [BoD04, ISO01] become cost elements important for adaptation decision making.

ix. *Monetary Costs:* It has been assumed in this research so far that VERSAG agents are co-operative and willingly supply capabilities to peers. Supplying capabilities to others is however an additional workload on agents and can affect their goals, and therefore supplier agents can charge the clients. Also, capabilities can be sold as licensed software that needs to be purchased before use. An agent can also incur a cost for using resources such as processing capacity and network bandwidth (e.g. Amazon EC2 Service[6]). In such situations, the various *monetary costs* need to be taken into account when the agent selects a set of capabilities to use.

x. *Status of Network:* The status and type of the network path over which a capability has to be acquired affects the desirability of the capability. Wireless network links are affected by higher loss rates, packet delays and jitter [NRT04], and therefore a capability accessible via a wireless network is generally less desirable than one accessible via a wired network. Both of these in turn are preferable over a capability on a remote network whose status is unknown. Therefore, we consider the status of network path over which capability acquisition occurs as a criterion to be used in capability selection.

xi. *Reuse Factor:* When an agent has a choice of multiple capabilities to select from, it may prefer to acquire one which has a higher possibility of being reused. For example, a capability that has additional functionality needed later in the itinerary, and supports multiple operating systems, is preferable

[6] http://aws.amazon.com/ec2/

when the situation does not impose limitations on agent size. Thus, we define *reuse factor* as a cost element for consideration in adaptation decision making.

We have identified eleven criteria discussed above that can be used as cost elements for consideration in capability selection. Techniques for *a priori* estimation of various cost components are presented later in this chapter. It should be noted that the above list of cost elements is not exhaustive. It is also noteworthy that most cost criteria have interrelationships which need to be clearly identified and managed in order to avoid duplication of cost factors. For example, energy consumption, computational needs, communication (related to network usage) and fidelity (related to precision) are closely related attributes of software applications [FlS99, KiF02].

It is neither necessary nor feasible to consider each one of these criteria before a decision making step due to the overheads involved in estimating their costs. Thus, it is necessary for the agent to be aware as to which criteria are relevant and also to consider their relative importance. Some cost criteria are implicit and come into effect due to the environment (e.g. memory/CPU limits of devices) while others are explicitly specified. These are specified as constraints and become part of the input to the decision making process.

Having introduced possible cost elements that can influence an agent's capability selection process, we proceed to describe our proposed solution. The next sub-section briefly describes how cost-efficient decision making is incorporated in a VERSAG agent as an adaptation feature.

4.3.3 Cost-Efficient Capability Acquisition as an Adaptable Feature

In the VERSAG agent reference architecture, the *adaptation service* was designated as the module with the know-how to decide on agent adaptation steps, and the *capability exchange service* was designated with the task of acquiring capabilities from remote agents. Both of these modules can be realised as capabilities and therefore can be dynamically added or replaced from an agent. This in turn allows a cost-efficient capability acquisition feature to be implemented as a set of co-operating capabilities and to be incorporated into a VERSAG agent. Details how this

has been realised in our VERSAG prototype are provided in chapter 5. The description of the feature in the current chapter is at a conceptual level and therefore is not concerned with identifying how functionality is distributed amongst capabilities, their boundaries or communication amongst capabilities.

4.3.4 Capability Acquisition Process

In the example scenario of sub-section 4.3.1, it was seen that the agent's itinerary activity requires capabilities matching three capability queries. Therefore, in order to carry out an activity, the agent has to first search peer agents for matching capabilities. Then, from amongst the located capabilities, the agent has to select the most cost-efficient set of capabilities, acquire them and update the itinerary to use them. We now present our proposed set of steps for achieving the goal of searching for and acquiring a set of capabilities matching these queries. These steps are further illustrated in the algorithm of Figure 4-2.

Step 1: For each capability query, search for matching capabilities in the agent's local repository and from peer agents. The search results are descriptions of matching capabilities that the agent already contains in its repository and those that are available from peers. Search protocols are implementation specific and consequently may be changed as needed.

Step 2: Combine received capability descriptions to build *capability groups* that can fulfil the given activity. If no groups can be formed, the activity has to fail. Group formation takes into account limitations of capability instances and their compatibility with each other.

Step 3: Identify the set of *constraints* that apply to the current activity. Some of these are explicitly specified by the user while others are generated as a result of environmental conditions sensed by the agent (through its context-awareness). Absolute constraints (see Definition 10 in section 3.3.2) place limits on values that cost elements can take while a single relative constraint (see Definition 9 in section 3.3.2) specifies the relative importance of cost elements. Cost elements not present in these constraints are considered to be irrelevant to the current activity.

Step 4: Use the *cost model* to sort the capability groups in increasing order of cost. The cost model makes estimates for relevant cost elements and uses them to sort the alternatives. This step is the heart of the cost-efficient capability acquisition feature and is described in detail in the next sub-section.

Step 5: Select the least cost capability group and acquire the relevant capability instances if they are not already with the agent. If acquisition fails, acquire the next best capability group. If a group could not be acquired, the activity fails.

Step 6: Update the agent itinerary by replacing capability query entries with the acquired capability instance details.

Algorithm: Capability acquisition algorithm

Input: $CQ = \{cq_1, cq_2, \ldots, cq_m\}$, set of capability queries needed

1 Search for capabilities matching CQ

2 Form capability groups $G = \{g_1, g_2, \ldots\}$ where each g_i can fulfil the activity
if $G \neq \emptyset$ **then**
Fail activity
end if

3 Build set of constraints $Cons = \{con_1, con_2, \ldots\}$ from explicit and implicit constraints identified through context sensing.

4 Apply cost model to select and sort *G* in increasing order of costs

5 **For each** g_i in G
Acquire capability instances of g_i by applying $get()$operation
If acquisition successful
Break loop
End if
End for
If a completed group is not available
Fail activity
End if

6 Update itinerary, replacing references to capability queries with references to corresponding capability instances that were acquired

Figure 4-2: Capability acquisition algorithm of an agent

The above algorithm assumes that the agent acquires all needed capabilities immediately prior to execution (i.e. follows a "light-travel" capability management policy) and therefore fails the activity if suitable capabilities could not be found. The purpose of grouping capabilities and selecting the least cost group as opposed to selecting individually "cheapest" capabilities is to identify and eliminate incompatible capability combinations early on in the selection process.

In our example scenario, the available capabilities identified from step one of the process were specified in Table 4-4. For group formation (step two), we assume there are no incompatibilities amongst the found capabilities. The set of all possible groups, *G* is as follows:

$$g_1 = \{c_{gnu-mysql@Dave}, c_{toadsql}, c_{searchV1}\}$$
$$g_2 = \{c_{gnu-mysql@Dave}, c_{frogsql}, c_{searchV1}\}$$
$$g_3 = \{c_{gnu-mysql@OSS}, c_{toadsql}, c_{searchV1}\}$$
$$g_4 = \{c_{gnu-mysql@OSS}, c_{frogsql}, c_{searchV1}\}$$
$$g_5 = \{c_{larry-mysql}, c_{toadsql}, c_{searchV1}\}$$
$$g_6 = \{c_{larry-mysql}, c_{frogsql}, c_{searchV1}\}$$

The results of step three are a relative constraint (con_1) and two absolute constraints (con_2, con_3) as follows.

$$con_1 = \{(time, 5), (network, 3), (cpu, 2), (precision, 2)\}$$
$$con_2 = (a_{DBsearch}, time, 0..5000)$$
$$con_3 = (a_{DBsearch}, precision, 5..10)$$

Step four of the algorithm identifies the need to estimate relevant cost components and aggregate them for each available alternative in order to compare between alternatives. We now proceed to develop a multi-criteria cost model to accomplish this step.

4.3.5 Cost Model

We now focus on the cost model which assesses the available alternatives (i.e. capability groups) with respect to the assessment criteria (i.e. cost constraints) in order to select the least cost alternative. Figure 4-3 presents an overview of this cost model. It takes a list of available capability groups and applicable constraints as input and generates a list of capability groups that meet the constraints and is sorted in increasing order of cost.

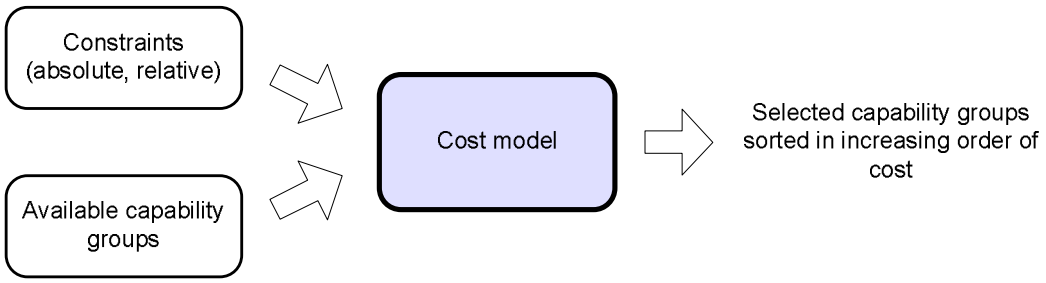


Figure 4-3: Overview of cost model for capability group selection

Each capability group contains details of a collection of capabilities that can fulfil the set activity, but not the actual capability instances. The capabilities themselves are acquired after the most suitable group has been selected. Constraints to be considered include those explicitly specified as well as implicit constraints due to environmental limitations. The relative importance of each cost element is described by the *relative constraint* (see Definition 9 in section 3.3.2). Any *absolute constraints* (see Definition 10 in section 3.3.2) specify permissible limits on cost elements. This process corresponds to step four of the capability acquisition algorithm described in Figure 4-2.

The functionality of the cost model could be divided into two sub-problems as follows:

a. From a given set of capability groups, select the groups that satisfy given absolute cost constraints. To satisfy an absolute constraint, the estimated value of the cost element applicable to that constraint must lie within the permissible values that the constraint specifies.
b. Order the selected groups in increasing order of their overall costs, where overall cost is calculated by combining multiple cost elements with differing levels of relative importance.

Solving the first sub-problem requires estimating the relevant cost values and comparing them against the values specified within the constraint. By doing this first, it is possible to eliminate alternatives that do not meet absolute criteria, thereby reducing the number of alternatives to consider in the second sub-problem.

In the second sub-problem, it is once again necessary to estimate the relevant cost values for each alternative. Once the cost estimates are available, we combine each alternative's cost values (expressed using different measures) to arrive at an overall cost value that can be used to compare the alternatives. This is a Multiple Criteria Decision Making problem (MCDM) [HwY81, Tri00]. Since the current problem has a finite number of alternatives to select from, it can be specifically classified as a Multiple Attribute Decision Making problem (as opposed to a Multiple Object Decision Making problem which has an infinite number of alternatives to select from) [HwY81].

An MCDM problem contains a number of *alternatives* which represent the choices available to the decision maker. A problem also has multiple associated *decision criteria* (or attributes) where the desirability of an alternative can be calculated based on each decision criterion. Criteria are of two types: those where desirability increases as the value increases are *profit* criteria (e.g. accuracy) and those where desirability increases as the value decreases are identified as *cost* criteria (e.g. time). An optimal solution to such a problem is an alternative which has the highest desirability for each criterion (i.e. maximises profit criteria and minimises cost criteria). However, if an MCDM scenario involves conflicting criteria (e.g. computational requirements and energy efficiency in the current situation) as is often the case, there is no feasible optimal solution. In such situations, the objective is to obtain a best possible solution. An MCDM problem can be represented using a decision matrix as shown below [page 2,3 of Tri00].

Let $A = \{a_1, a_2, \dots, a_m\}$ represent m finite alternatives, and $C = \{c_1, c_2, \dots, c_n\}$ represent *n* criteria based on which each alternative is evaluated. Then, a decision matrix is an $(m \times n)$ matrix D where element $x_{i,j}$ represents the desirability of alternative a_i in terms of criterion c_j.

$$D_{m\times n} = \begin{array}{c} \\ \boldsymbol{a_1} \\ \boldsymbol{a_2} \\ \dots \\ \boldsymbol{a_m} \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{c_1} & \boldsymbol{c_2} & \dots & \boldsymbol{c_n} \end{array} \\ \begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} \\ \dots & \dots & \dots & \dots \\ x_{m,1} & x_{m,2} & \dots & x_{m,n} \end{bmatrix} \end{array} \tag{4.1}$$

Thus, each row in the matrix represents an alternative a_i while a column represents a criterion c_j.

Triantaphyllou and Mann [TrM89] identify three steps in solving an MCDM problem. They are:

a. Identify the available alternatives and criteria relevant to the decision making
b. Assign weights to the criteria indicating their relative importance
c. Calculate overall preference for each alternative

In the current cost model, the first two steps have been already performed with the capability groups representing the alternatives and the relative constraint identifying the criteria as well as assigning weights to them. Thus, the task of the cost model is to perform the third task of calculating preferences for each alternative.

Many methods such as the Analytic Hierarchy Process (AHP), Weighted Sum Model (WSM), Weighted Product Model (WPM), and the Evidential Reasoning (ER) approach have been proposed and are used to solve MCDM problems [HwY81, Saa90, Tri00, TrM89]. In this thesis, we select the Analytic Hierarchy Process (AHP), which has been widely used [Saa90]. We provide a brief description of the AHP next.

The Analytic Hierarchy Process [Saa77, Saa90] first decomposes a given problem into a hierarchy with the goal at the top, selection criteria in the middle-tier(s) and the available alternatives in the bottom-tier. Multiple middle-tiers are used when the criteria themselves are made up of sub-criteria. This is not required in our current problem. A hierarchy, as applicable to the current cost model is shown in Figure 4-4. Each element in the hierarchy is assigned a normalized *priority value* which indicates its relative importance in comparison to the other elements at the same level. The goal which is at the top of the hierarchy has priority value 1. AHP recommends assigning priorities for criteria in the middle-tiers using pairwise comparisons [Saa90]. Since the cost model receives priorities (weights) for criteria in the form of a relative constraint, we do not discuss it here. Pairwise comparison is introduced in sub-section 4.4.1 when we discuss approaches to capture user-specified inputs.

Arriving at priority values for the alternatives (bottom-tier) is the aim of the selection process. Since the values indicate priority, the alternative (i.e. capability group) with highest priority value has the least cost.

For calculating the priorities of alternatives, AHP too uses a decision matrix. In this matrix, element $x_{i,j}$ represents the relative desirability of alternative a_i in terms of criterion c_j. Since criteria are expressed using different measures, in order to enable comparison between them, columns of the matrix are normalized by dividing each element by the column sum [HwY81]. The normalization is done such that $\sum_{i=1}^{m} x_{i,j} = 1$. The priorities (i.e. relative importance) of criteria are represented in another similarly normalized column vector. By multiplying the decision matrix with this vector, priority values for the alternatives are calculated. This relationship is given as follows in [Tri00].

$$A_{AHP-best} = \max_i \sum_{j=1}^{n} x_{i,j} \times w_j \quad \text{for } i = 1,2,3,\ldots,m. \tag{4.2}$$

where, w_j represents the priority of criterion c_j.

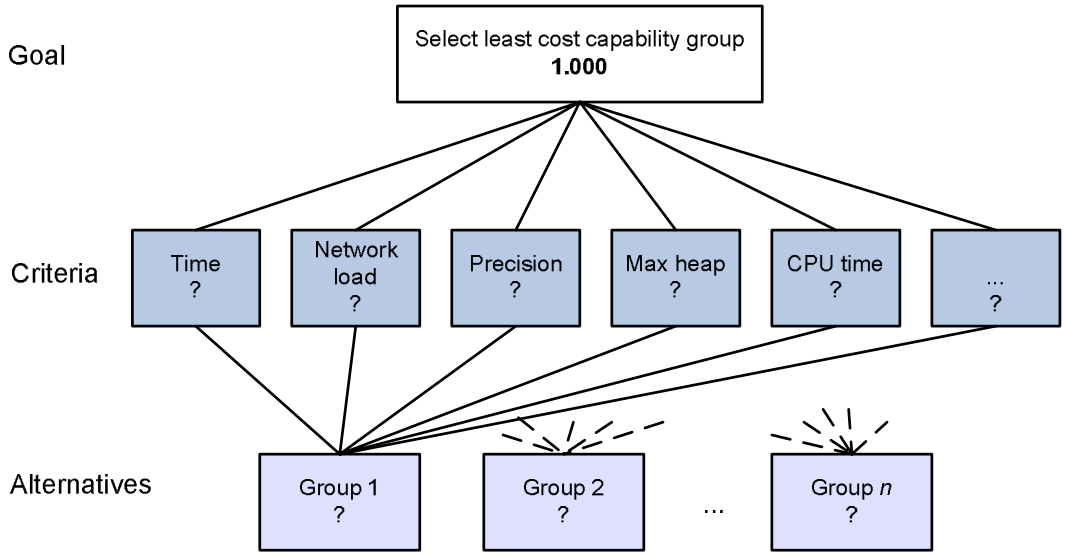


Figure 4-4: Selection process structured as an AHP hierarchy

An algorithm for the cost model, combining the two sub-problems presented previously is presented in Figure 4-5.

Algorithm: Cost model algorithm

Input: Set of cost elements $Q = \{q_1, q_2, \dots, q_k\}$
Relative constraint $\{(q_1, w_1), (q_2, w_2), \dots, (q_m, w_m)\}$ where $m \leq k$
Absolute constraints $Cons = \{con_1, con_2, \dots, con_n\}$ where $n \leq k$
Alternatives $G = \{g_1, g_2, \dots, g_p\}$

Output: G sorted in decreasing order of utility

1 **For each** con_i in $Cons$ with cost element q_i
  For each group g_j in G
    Estimate cost in terms of q_i for group g_j
    If estimated cost does not satisfy con_i
      Remove group g_j from G
    End if
  End for
End for
If $G = \emptyset$
  Fail activity
End if

2 Construct vector $P_{m\times 1} = [w_1 \quad w_2 \quad \dots \quad w_m]^T$ from relative constraint
Normalize $P_{m\times 1}$ such that $\sum_{i=1}^{m} w_i = 1$

3 **For each** group g_i in G
  For each cost element q_j in the relative constraint
    If cost estimate of g_i not available
      Estimate cost in terms of q_j for group g_i
    End if
  End for
End for

4 Invert cost estimates where necessary to get corresponding profit (utility) values
Construct decision matrix $U_{p\times m}$ from estimated utility values
Normalize $U_{p\times m}$ column wise such that $\sum_{i=1}^{p} u_{i,j} = 1$ for each *j*

5 Calculate utility vector $V_{p\times 1} \leftarrow U_{p\times m} \times P_{m\times 1}$
Sort $V_{p\times 1}$ in decreasing order of utility

Figure 4-5: Multiple criteria cost aggregation algorithm

We further describe the algorithm illustrating how it is used with our example scenario. The hypothetical cost estimates given in Table 4-5 are assumed for the six alternative groups identified earlier. CPU usage and accuracy are expressed in a 10-point scale [page 27 of HwY81] where higher numbers indicate a higher value (e.g. maximum CPU usage is 10).

Table 4-5: Simulated cost estimates for alternatives

Group	Time (ms)	Network load (MB)	CPU usage	Accuracy
g_1	1650	2.8	5	6
g_2	1950	3.4	6	8
g_3	1250	1.8	4	8
g_4	950	1.2	4	4
g_5	3520	0.9	5	9
g_6	1925	2.1	7	7

Inputs to the cost model are as follows.

$$Q = \{time, network\ load, CPU, memory, accuracy\}$$
$$con^R = \{5, 3, 2, 2\}$$
$$R = \{(a_{search_{DB}}, time, 0..5000), (a_{search_{DB}}, accuracy, 5..10)\}$$
$$G = \{g_1, g, g_3, g_4, g_5, g_6\}$$

Step 1: Evaluate each alternative (i.e. capability group) against the *absolute constraints*. The group cost in terms of the relevant cost element associated with the constraint has to be estimated. Groups that fail to meet absolute constraints are removed. If no groups remain at the end of this step, the process fails. Only costs required to evaluate the absolute constraints are estimated at this step.

- The constraint $(a_{search_{DB}}, time, 0..5000)$ is satisfied by all groups.
- For $(a_{search_{DB}}, accuracy, 5..10)$, accuracy of g_4 is 4 (< 5) and it fails the constraint. Therefore g_4 is removed from available alternatives.

Step 2: Construct a column vector of priorities from the relative weights in the *relative constraint*. The vector is normalized by dividing each element with the column sum to obtain comparable scales.

- We normalize $[5 \quad 3 \quad 2 \quad 2]^T$ to get,

$$P_{4\times1} = \begin{bmatrix} 0.42 \\ 0.25 \\ 0.17 \\ 0.17 \end{bmatrix}$$

Step 3: For the remaining alternatives (i.e. capability groups), estimate costs for those elements that are part of the relative constraints. For example, if a cost element has an associated priority value of 0, then it is not relevant and costs need not be estimated. These values are already provided in Table 4-5.

Step 4: Construct a decision matrix from the group cost estimates with rows representing alternatives and columns representing cost elements.

- Rows represent alternatives g_1, g_2, g_3, g_5, g_6. Of the four criteria, since *time, network load* and *CPU* represent cost values, the values are inverted to convert into utility values. The values for *accuracy* already indicate a utility.

$$U_{5\times4} = \begin{array}{c} \\ \boldsymbol{g_1} \\ \boldsymbol{g_2} \\ \boldsymbol{g_3} \\ \boldsymbol{g_5} \\ \boldsymbol{g_6} \end{array} \begin{array}{c} \boldsymbol{C_{time}} \quad \boldsymbol{C_{nw}} \quad \boldsymbol{C_{cpu}} \quad \boldsymbol{C_{pr}} \\ \begin{bmatrix} 1/1650 & 1/2.8 & 1/5 & 6 \\ 1/1950 & 1/3.4 & 1/6 & 8 \\ 1/1250 & 1/1.8 & 1/4 & 8 \\ 1/3520 & 1/0.9 & 1/5 & 9 \\ 1/3520 & 1/2.1 & 1/7 & 7 \end{bmatrix} \end{array}$$

- Since each column uses a different unit for measurement, each element values is divided by the sum of its column to normalize the matrix. The decision matrix, after normalizing is:

$$U_{5\times4} = \begin{bmatrix} 0.24 & 0.13 & 0.21 & 0.16 \\ 0.21 & 0.11 & 0.17 & 0.21 \\ 0.32 & 0.20 & 0.26 & 0.21 \\ 0.11 & 0.40 & 0.21 & 0.24 \\ 0.11 & 0.17 & 0.15 & 0.18 \end{bmatrix}$$

Step 5: Calculate a utility vector *V* by multiplying the decision matrix *U* with the priority vector *P*.

- Multiply $U_{5\times4}$ with the priority vector $P_{4\times1}$

$$V = \begin{bmatrix} 0.24 & 0.13 & 0.21 & 0.16 \\ 0.21 & 0.11 & 0.17 & 0.21 \\ 0.32 & 0.20 & 0.26 & 0.21 \\ 0.11 & 0.40 & 0.21 & 0.24 \\ 0.11 & 0.17 & 0.15 & 0.18 \end{bmatrix} \times \begin{bmatrix} 0.42 \\ 0.25 \\ 0.17 \\ 0.17 \end{bmatrix}$$

$$= \begin{matrix} \boldsymbol{g_1} \\ \boldsymbol{g_2} \\ \boldsymbol{g_3} \\ \boldsymbol{g_5} \\ \boldsymbol{g_6} \end{matrix} \begin{bmatrix} 0.19 \\ 0.18 \\ \mathbf{0.26} \\ 0.22 \\ 0.15 \end{bmatrix}$$

- The groups in decreasing order of utility are: $g_3 > g_5 > g_1 > g_2 > g_6$. The alternative with highest utility, g_3 is the alternative with least cost.

The cost model produces a list of capability groups sorted in decreasing order of cost. This list is then used by the capability acquisition process (described in the previous sub-section) to select the group of capabilities with least cost to fulfil the agent's task.

Having presented our approach to cost-efficient capability acquisition to illustrate adaptation decision making in VERSAG, in the next section we proceed to describe techniques for *a priori* estimation of cost elements which make up the decision making criteria.

4.4 Estimating Cost Elements

Our approach to cost-efficient decision making described thus far in this chapter is dependent on being able to compute cost estimates for the various relevant criteria. Therefore, in this section we focus on techniques to generate such cost estimates. While the cost model supports any number of cost criteria, we limit our discussion on cost estimating techniques to the key cost elements given in Table 4-6. Time and network load are the primary cost elements with others treated as secondary criteria since these are the factors that mobile agent performance optimisation research typically focuses on [BrR04, CGK05, Pic01].

Table 4-6: Overview of supported cost elements

	Element	Unit/Range	Description
1.	Time	Milliseconds	Time taken to fulfil a given *itinerary step*.
2.	Network load	Bytes	Total network traffic generated when fulfilling a given *itinerary step*.
3.	Memory	0-10	A developer assigned ranking indicating the maximum memory requirement of a capability.
4.	CPU usage	0-10	A developer assigned ranking of the CPU intensity of a capability.
5.	Accuracy	0-10	A developer assigned numerical indication of how accurate the capability's generated output is.

Two primary approaches taken to generate cost values are: computing using pre-established formulae and using user provided estimates. Examples of both these approaches are presented in this section. This section first looks at how user specified inputs can be captured and then proceeds to describe techniques and formulae to estimate cost elements.

4.4.1 Capturing User Specified Inputs

While the capability acquisition feature aims to enable agents to autonomously make decisions without requiring user intervention, it is necessary to get some user input to assist the decision making. Such user specified input can be classified into two classes based on their intended use:

a. To specify relative importance (weight) of cost criteria
b. To help calculate cost of given criteria

It was earlier mentioned that the relative importance of cost criteria used for decision making are formally specified in a *relative constraint*. It is expected that the relative constraint will be constructed by the agent based on input received from the user. User provided cost elements tend to be qualitative (or fuzzy) [HwY81] and need to be transformed into quantitative values before they are usable in the cost model .

The analytic hierarchy process (AHP) proposes the use of *pairwise comparisons* to capture these relative weights. This process involves a user comparing the available alternatives (i.e. cost criteria in this case) in pairs, specifying the relative importance of one criterion in comparison to the other. The fundamental scale defined by Saaty [Saa08] (shown in Figure 4-6) can be used to compare the criteria linguistically and to quantify them [page 24 ofTri00]. For example, Bob may specify that "time is strongly important than network load", which quantifies to a value of 5 on the scale. Similar comparisons are elicited for all relevant criteria and then combined to form a pairwise comparison matrix from which relative importance of the multiple criteria can be calculated. While widely used, this method suffers from possible inconsistencies when combining the pairwise comparisons. It is also not appropriate when a large number of criteria are present due to the high number of comparisons required [page 25 ofTri00].

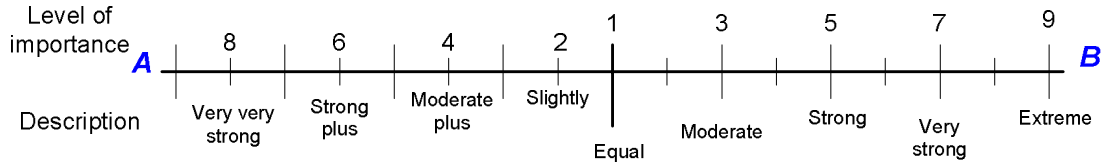


Figure 4-6: The fundamental scale to compare two alternatives (source [Saa08])

In the Multi-Attribute Resource Intermediary (MARI) project, Tewari and Maes [TeM01] automatically infer relative weights for different attributes depending on the user's specifications for the individual criteria. They assume that if a user accepts a large range of values for a criterion it is of low importance. As the accepted range of values becomes narrower, the importance of the criterion increases. The CRITIC method [DMP95] also does away with the need of human input and calculates objective weights for different criteria using statistical techniques.

In this research we do not select any specific technique for capturing user preferences. Instead, we expect that different techniques could be used by agents depending on the circumstances. In our implementation, we assume direct input of the relative weights by a user. In [Eck68], the approach of direct rating has been identified as being comparative to other approaches, including pairwise comparison.

The second type of user-specified inputs is those that help calculate cost estimates. Since the cost criteria are used to quantify capabilities developed by external parties, the agent requires certain data to be provided (as part of capability meta-data) about them to aid in estimating cost values. Examples for such data include: resource requirements, accuracy of generated results, reliability, and speed of execution. While the developer of a capability is the most suitable party to provide such data, because the cost model needs to compare capabilities developed by multiple parties it is necessary to take steps to ensure that data provided are consistent and comparable. We envision that this could be achieved by the introduction of a third party certification body (similar to Symbian Signed[7], Java Verified[8]). Responsibilities of such a body would include defining standard test environments in which capabilities are to be tested; and testing/certifying capabilities for quality and trust. Further investigation of this option is beyond the scope of this thesis.

We describe next, methods to estimate these criteria. These cost estimates are then used in the proposed cost model to calculate costs of each alternative.

4.4.2 Estimating Time

Time, being a key quality metric in many mobile/distributed applications is investigated first. The time taken to execute an itinerary step constitutes several components. First, the agent has to search for capability instances, group the results and select the least cost alternative from amongst them. Then, it has to acquire the selected capability instances and execute them. The time spent on decision making (i.e. searching for, grouping and selecting an alternative) is present irrespective of which alternative is selected. For continuously running capabilities (i.e. *cyclic* capabilities), running time is omitted.

Given that a capability group consists of n capabilities, the time (T^{step}) to sequentially acquire the capabilities and fulfil the itinerary step by sequentially executing these capabilities can be expressed as follows:

---

[7] http://www.symbiansigned.com

[8] http://javaverified.com

$$T^{step} = T^{search} + \sum_{i=1}^{n} \left(T_i^{cap\_req} + T_i^{cap\_res} + T_i^{start\_p} + Tp_i^{loc} \right) \quad (4.3)$$

where,

T^{search} is time to search and select capabilities,

$T_i^{cap\_req}$ is time to request capability i,

$T_i^{cap\_res}$ is time to receive capability i in response,

$T_i^{start\_p}$ is time to start execution of capability i, and

Tp_i^{loc} is time to execute capability i at location loc.

Time taken to communicate over a network is proportional to the size of data being sent and inversely proportional to the bandwidth of the network. Thus, time T to move B bytes over a network link with latency λ and bandwidth bw can be represented as:

$$T = \lambda + \frac{B}{bw} \quad (4.4)$$

The latency and bandwidth of the network path over which capability i is requested and acquired are λ_i and bw_i, respectively. From Equation 4.4,

$$T_i^{cap\_req} = \lambda_i + \frac{(B^{creq} + B^{ovrhd})}{bw_i}$$

$$T_i^{cap\_res} = \lambda_i + \frac{(B_i^{cap} + B^{ovrhd})}{bw_i}$$

where,

B^{creq} is the size of any capability request,

B_i^{cap} is the size of the capability i and

B^{ovrhd} is the overhead associated with a single request or response.

Applying these to equation 4.3 yields,

$$T^{step} = T^{search} + \sum_{i=1}^{n} \left(\lambda_i + \frac{(B^{creq} + B^{ovrhd})}{bw_i} + \lambda_i + \frac{(B_i{}^{cap} + B^{ovrhd})}{bw_i} + T_i^{start\_p} + Tp_i^{loc} \right)$$

$$T^{step} = T^{search} + \sum_{i=1}^{n} \left(2\lambda_i + \frac{(B_i{}^{cap} + B^{creq} + 2B^{ovrhd})}{bw_i} + T_i^{start\_p} + Tp_i^{loc} \right) \quad (4.5)$$

If the bandwidth and latency are the same over all network links, this leads to,

$$T^{step} = T^{search} + 2n\lambda + \frac{n}{bw}(B^{creq} + 2B^{ovrhd}) + \sum_{i=1}^{n} \left(\frac{B_i{}^{cap}}{bw} + T_i^{start\_p} + Tp_i^{loc} \right) \quad (4.6)$$

Of the variables in Equation 4.5, the number of capabilities (n) is known by the agent and the size of each capability (B_i^{cap}) is known from the capability details in search results. Network parameters (λ_i and bw_i) are expected to be obtainable either from the agent's context-sensing capability or from an external context service. The size of a capability request (B^{creq}) is also fixed for a particular capability exchange protocol and therefore known to the agent.

The request (or response) overhead involved (B^{ovrhd}) is dependent on network conditions as well as the capability exchange protocol and data size. We assume it to be a fixed value for a given estimation and to be available to the agent. The context-awareness module of an agent is a candidate for monitoring the network characteristics and determining this value.

The time to search for and select a set of capabilities (T^{search}) is dependent on network conditions and protocols used. It is assumed to be fixed for a given agent and network setup and therefore available to the agent.

4.4.2.1 Capability Start up and Execution Time

To estimate time to fulfil an itinerary step (T^{step}), it is necessary to be able to predict how long start up ($T_i^{start\_p}$) and execution (Tp_i^{loc}) of individual capabilities will

take. While VERSAG assumes sequential execution of capabilities, certain capabilities are *passive* and do not have an active process (i.e. provide methods for use by other capabilities or the agent kernel). There are also capabilities that continuously execute as background processes allowing the agent to execute other capabilities in the foreground. For these two types, for the sake of simplicity, execution time is considered nil, ignoring any effect that they may have on the execution of other capabilities. For capabilities that need to sequentially run to completion, it is necessary to be able to predict execution time. Execution time could range from a few milliseconds to hours or even longer depending on different factors. We group these factors as follows:

- *Capability related factors:* functionality to carry out, algorithms used, program implementation details
- *Location related factors:* available resources (e.g. CPU, memory, input/output devices), resource scheduling, other active processes
- *Data related factors:* size of data, nature of data (decides the complexity of processing)

Estimating program execution time is a non-trivial task and an active area of research [Din02, KLZ03, SFT98]. The most common technique of program run-time prediction is to identify programs that are "similar" to the current one from historical data and to use their actual run-times to generate a prediction for the current one [KLZ03, SFT98]. This approach however is not suitable for VERSAG due to the following reasons:

- The prediction process itself is complex and can result in decision making time (T^{search}) being significant in comparison to the agent's itinerary execution time, thus making the extra overhead unacceptable.
- VERSAG's diverse and heterogeneous range of application capabilities and diverse platforms makes it difficult to collect sufficient historical data. Such a collection of historical data, if available, would be too large to be stored within the agent. If it is stored remotely, this then adds an external dependency and extra communication overhead to the overall process.

- VERSAG uses predicted program (i.e. capability) run-times to compare different alternatives implementing similar/identical functionality. These alternatives are likely to have similar past programs, which in turn would lead to similar predicted runtimes. This therefore results in these predictions being unable to distinguish differences between the various capabilities.

Another approach to application run-time prediction is to ask users (i.e. program creators) for guidance on expected run-times [SFT98]. We propose a similar approach which requires capability developers to provide a run-time estimate for each capability.

A capability developer is therefore required to execute the capability in a standard test environment and report the observed execution times as part of its meta-data. Furthermore, this information must indicate the relationship execution time has with factors such as data and computational resources. The agent is then able to use these properties to calculate an estimated capability execution time for its current itinerary step. In this situation, a third party certification body as described previously is required to ensure consistency of estimates across multiple capabilities.

The time to start a capability ($T_i^{start\_p}$) can be non-negligible depending on the capability's start up procedures and is therefore included as a separate component in the formula. For example, the time to start a capability is increased if it launches a GUI (Graphical User Interface) at start. It is dependent on both the nature of the capability and resources available at the particular location. Therefore, the capability developer is required to include an indication of capability start up time into its meta-data, allowing the agent to build an estimate taking into account available resources.

VERSAG thus depends on start up and execution time estimates provided by capability developers and included in the capability's meta-data for its estimation process.

4.4.3 Estimating Network Load

Network traffic generated when executing an itinerary is investigated as the second key cost criterion. Total network load generated during an itinerary step consists of

the load due to execution of the capabilities as well as load due to search and acquisition of capabilities. As with time estimates, the network cost of decision making (i.e. searching for suitable capabilities) is common for all the alternatives.

Given that an alternative has *n* capabilities, the network cost can be expressed as follows:

$$B^{total} = B^{search} + \sum_{i=1}^{n} (B_i^{acq} + B_i^{exec}) \tag{4.7}$$

where,

B^{search} is load generated during decision making,
B_i^{acq} is load generated due to capability acquisition and
B_i^{exec} is load generated while the capability executes.

The network load due to acquisition of a capability (B_i^{acq}) is made up of the request size (B^{creq}), capability size (B_i^{cap}) and the overhead of each request (B^{ovrhd}) and is shown in Equation 4.8 below. These values are available to the agent as mentioned in section 4.4.2.

$$B_i^{acq} = B^{creq} + B_i^{cap} + 2B^{ovrhd} \tag{4.8}$$

The network cost of capability execution (B_i^{exec}) is dependent on the nature of functionality and implementation specific details. Similar to capability execution time, a wide array of costs are expected here, ranging from no network usage to constant high volume network requirements. Therefore, the capability developer is required to indicate the capability's network traffic generation behaviour in its meta-data. Once again, this would be a simple specification expressing the load behaviour.

The network load due to capability search queries and responses (B^{search}) is dependent on the network conditions, protocols used and number of requests/responses. It is however relatively small when compared to the capability acquisition costs and is assumed to be a constant value known to the agent.

For *cyclic* capabilities, the load generated at runtime is omitted in our current estimates. We note however that if a cyclic capability generates a significant amount of network traffic, this could affect the accuracy of the selection process.

4.4.4 Estimating Computational Resources

Maximum memory requirement and CPU usage are two computational resources that are investigated as secondary cost elements in the current research. Due to differences in hardware and software configurations, an estimate of a capability's resource requirements for one device platform is generally not meaningful on other types of devices [SRD09]. The wide array of possible device configurations also makes it difficult to provide estimates for all possible devices.

Shimizu et al. [SRD09] propose an approach for predicting application resource requirements in an application-agnostic manner for previously unseen platforms. Their approach builds application profiles based on previous resource usage characteristics, which are then used to predict resource needs for new platforms. While such an approach could be used with the VERSAG cost model, the maintenance of a growing collection of past resource usage data does present an additional overhead in terms of capability size.

Therefore, VERSAG uses a ranking system which rates a capability based on its CPU usage or maximum memory requirements. A 10-point scale [page 27 of HwY81] is used with higher scale values indicating higher resource needs (i.e. costs). Thus, this proposed resource requirement rating is a constant value for a given capability. This approach significantly simplifies cost aggregation as the rating can be directly used as the cost estimate of a capability.

The use of a rating has the drawback that it does not give an indication of the actual memory or CPU requirements of a capability, making it impossible to verify against absolute constraints. CPU and memory ratings to capabilities are assigned by capability developers as part of a capability's meta-data. It is expected that mapping memory values on different platforms to rating points will be part of the third-party certification process described in sub-section 4.4.1.

Since a rating is given for an individual capability, it is necessary to aggregate these ratings to estimate a rating for an alternative (i.e. capability group). For a given group of capabilities, the memory/CPU rating is taken as the highest rating from among the capabilities within the group. That is, given a set of *n* capabilities, with resource cost ratings $c_1, c_2, \ldots c_i \ldots, c_n$ where $c_i \in [0..10]$, the cumulative cost rating of the set of capabilities is:

$$c_{group} = \max(c_1, c_2, \ldots, c_n) \tag{4.9}$$

4.4.5 Estimating Accuracy of Output

One objective of VERSAG is that capabilities should be software components reused and shared amongst agents. Thus, a capability can be used in applications which have different accuracy requirements. The cost element of accuracy is useful in such situations for agents to decide whether a given capability has adequate accuracy for its needs.

A 10-point scale, with higher ratings indicating higher accuracy is used as a measure of this criterion. The rating is to be assigned as part of the capability certification process and is not investigated in this thesis. The accuracy rating represents a desirable feature and therefore is directly used in the cost model as a utility value.

The overall accuracy when multiple capabilities are combined is assumed to be the lowest accuracy of the constituent capabilities. Thus, given a set of *n* capabilities, with accuracies $acc_1, acc_2, \ldots, acc_n$ the cumulative accuracy of the outcome of executing the set of capabilities is:

$$acce_{group} = \min(acc_1, acc_2, \ldots, acc_n) \tag{4.10}$$

4.5 Extensibility of the Cost Model

The cost model described in this chapter is primarily used by an agent to select capability instances when it has multiple alternatives with which it can achieve a given task. However, the cost model could be applied to a wider range of decision making steps in a VERSAG agent's life cycle. Examples include:

- Depending on environmental conditions and outcomes of its past actions, an agent sometimes needs to modify its future activities. By modelling these alternatives as sequences of capability executions, the cost model could be used for this purpose as it is.
- By extending the cost model to calculate cost estimates for agent migrations, it could be used to compare between alternative migration paths for an agent.
- The cost model could also be extended for use in the itinerary generation process (see 4.2.2) to select a suitable capability management policy for an agent's itinerary.

Furthermore, the cost model could be used at application design time to predict the costs of using VERSAG agents for a particular task compared to using other solutions.

4.6 Summary

The focus of this chapter was on how VERSAG agents can dynamically adapt to gain sophisticated features through acquisition of appropriate capabilities. Specifically, the question of how to make cost-efficient capability selection decisions from multiple alternatives was addressed. It was identified that multiple (possibly conflicting) criteria need to be considered in order to make the best decision. A multi-criteria decision making approach based on the Analytic Hierarchy Process was proposed and presented. The approach requires priorities of the various cost criteria and available alternatives to be given as input. It first estimates possible costs of the alternatives and then combines them to arrive at an overall least cost set of capabilities.

The chapter also identified possible cost criteria for use in this decision making process and described techniques for *a priori* estimation of several key criteria. It was also outlined that the proposed cost model could be extended for use with other decision making steps of VERSAG, both at run time as well as at design time.

A key issue in the proposed cost-efficient capability acquisition feature is to ensure that the overheads of *a priori* cost estimation and decision making do not outweigh

the benefits achieved. It is therefore a key evaluation question and in chapter 6, we describe experimental evaluations to demonstrate that this is indeed the case. Furthermore, chapter 6 presents experimental validations of the effectiveness of using our decision making strategy combined with the cost model for improving VERSAG performance. Prior to that, in the next chapter we present details of our VERSAG prototype implementation, which is the platform for demonstrating feasibility and conducting experimental evaluations of the concepts that underpin the proposed VERSAG framework.

C H A P T E R F I V E

5 VERSAG Implementation

Chapters 3 and 4 introduced and discussed the proposed VERSAG agent framework and its underlying theoretical concepts. This chapter details our implementation of the VERSAG agent framework.

5.1 Introduction

In the preceding chapters we presented the theoretical underpinnings of our proposed VERSAG agent framework as a novel enabling platform to build smart pervasive applications using compositionally adaptive mobile software agents. The formal foundations of the framework and the reference architecture of a VERSAG agent were described in chapter 3. It also described how new functionality can be incrementally added on to a basic itinerant agent with the aid of reusable software components termed *capabilities*. Chapter 4 proposed and discussed a multi-criteria cost model that allows agents to make adaptation decisions in a cost efficient manner based on environmental conditions/constraints and user preferences.

This chapter describes our prototype implementation of the VERSAG agent framework. The implementation was carried out with the aims of:

- Demonstrating the feasibility of the dynamic compositional adaptation proposed in this thesis for agents to acquire and discard capabilities during execution/runtime; and
- Experimentally validating the concepts proposed in the preceding chapters.

The prototype was built with 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 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*. This avoids creating yet another mobile agent toolkit [CPV07] and enables using agent services provided by the underlying toolkit. Also, this allows VERSAG agents to be introduced to existing multi-agent environments with minimum disruption.[9]
- The prototype should be able to run on diverse hardware and software platforms, the likes of which are commonly encountered by pervasive computing applications.

The rest of this chapter is organized as follows. Section 5.2 describes the selection of development platforms and tools for the prototype. In section 5.3 we describe the important decision steps involved in selecting a capability model for the prototype implementation. Section 5.4 then presents design and implementation details of the prototype. Information pertaining to the operation of the prototype is provided in section 5.5. We conclude this chapter in section 5.6.

5.2 Development Platform and Tools

This section describes the development platforms and tools used to build the VERSAG prototype.

Java [AGH05] is the language of choice for mobile agent programmers, and most current agent toolkits are built using it [ASF10]. Programs written in Java are compiled to an intermediate bytecode format that is targeted towards a virtual

[9] VERSAG is expected to be agent-platform agnostic, and it should be possible to have agents execute on multiple agent platforms. Realising this is however beyond the scope of the current implementation.

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. Therefore, Java has been selected as the implementation language for our VERSAG prototype. This choice is also aligned with the objective of building a platform-independent prototype.

Another objective of ours is to build the prototype using a current agent toolkit, and accordingly the JADE (Java Agent DEvelopment Framework) [BCP03] agent toolkit was chosen as the basis for VERSAG. While there have been many software agent toolkits developed over time, only a few of them such as JADE, Aglets [LaO98, LOK97], Voyager [Voy07], Tryllian [Try07], JACK [HRH01] and Tracy [BMS05] are currently available. While JACK does not support agent mobility, Tryllian and Tracy are no longer actively maintained. JADE on the other hand, is an open source project maintained by Telecom Italia and is one of the most popular agent platforms at present. It is compliant with FIPA [Fip10] 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. A brief introduction to the key elements of a JADE agent platform is provided next.

5.2.1 Key Elements of JADE

A distributed JADE agent platform consists of multiple *containers*. A container runs in a single Java Virtual Machine and can host multiple agents. In this research, a JADE container is analogous to a VERSAG *location*. A special container named the *main container,* which has to be started first, manages the platform and other containers register with it to join the platform. The main container hosts a centralized agent management system (AMS agent) and a Yellow Pages directory service (DF agent). It also maintains a registry of all peripheral containers belonging to the platform and their transport addresses. With agent messaging handled by the platform, agents can be addressed by their name without the need to provide transport addresses.

It is possible for different JADE agent platforms to communicate and have agents migrate amongst them (i.e. inter-platform migration). However, there are no

discovery mechanisms between them and communication is slower. Since issues related to agent discovery and communication mechanisms are not the focus of the current research, we limit our implementation and experiments to a single agent platform. In Figure 5-1 we illustrate a distributed JADE platform consisting of three interconnected JADE *containers*. While the example shows one container per computer, it is possible to run multiple containers on a single computer.

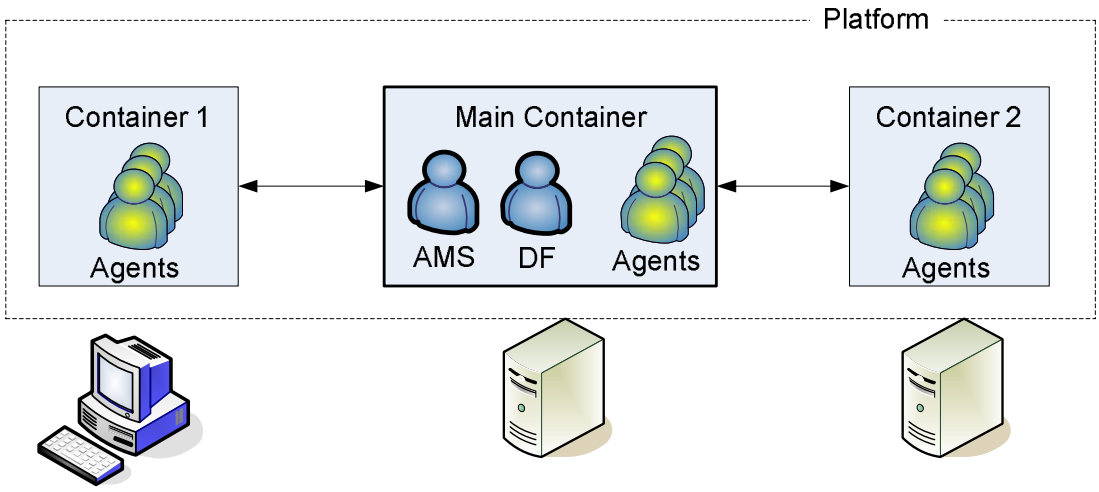


Figure 5-1: Key architectural elements of a JADE platform (adapted from [BCG07])

We presented above, a concise overview of the key elements of JADE which are necessary for our discussion in the remainder of this chapter and in chapter 6. Further details of JADE can be found in [BCP03, BCT07, BCG07] and the JADE web site [Jad09]. The next section describes the important process of choosing a capability model for the prototype implementation.

5.3 Capability Model Implementation

A capability, identified as a key element of the VERSAG approach, is essentially a reusable software component [CoH01]. A VERSAG agent compositionally adapts at runtime by acquiring and discarding capabilities, thereby changing its internal structure. An agent thus gains advanced features such as context-awareness, self-adaptability, and application-specific behaviours by executing capabilities. Details on how capabilities interact with other software elements and the runtime environment required to support their execution are defined in a *capability model* (analogous to a

component model as defined in [CoH01, WeS01]) and the set of software entities required to support execution of capabilities is a *capability model implementation* (analogous to a component model implementation in [CoH01, WeS01] or a container in [LaW07]). This sub-section describes the choice of a capability model implementation for use with our prototype and justifies its selection.

In chapter 3 we presented requirements and desirable features of a capability model for VERSAG. We recapture them below:

- *Life cycle*. There should be a standard mechanism for an agent to execute multiple capabilities simultaneously, start/stop a capability without adversely affecting the agent's functioning or that of other independent capabilities executing on the agent.
- *Transportable/portable*. Capabilities should be packaged in a manner such that they are transportable over a computer network and be portable across multiple platforms.
- *Interfaces*. There should be a well-defined and minimal interface which allows third parties to develop capabilities and define custom interfaces as needed.
- *Communication*. A capability should be able to communicate with other capabilities when needed and to access services provided by the agent.
- *Naming and meta-data*. It should be possible to describe a capability with associated information such as a unique identifier, implemented functionality, input/output parameters and so on.
- *Evolution support*. Evolution support, allowing different versions of a capability to co-exist on an agent is a desirable feature of a capability model.
- *Standards*. A capability model underpinned by standards would allow capabilities to be reused in environments beyond VERSAG agents and allow components from a wider environment to be used in VERSAG.

Previous works on compositionally adaptive agents, as seen in chapter 2, have preferred to develop their own component models. While a custom-developed capability model would lead to better compliance with requirements, it would require

reimplementation of many features already available in other component models. It is also unlikely to meet the requirement of being compatible with a widely used standard. This research therefore investigated the possibility of reusing an existing component model for defining VERSAG capabilities. Three possibilities considered are as follows.

- *JADE:* The JADE agent toolkit [BCP03] allows dynamic loading of functionality on to agents. A JADE agent's functionality is represented as programmer developed *Behaviours* (`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` is capable of receiving new *Behaviours* over incoming ACL (Agent Commuication Language) messages. The code of the new *Behaviour* should be included in the message. At the recipient agent, the new *Behaviour* is added to the existing pool of *Behaviours* and scheduled with them. This approach, if chosen would result in the JADE agent behaviour life cycle, and interface becoming that of the capability model. As there is no in-built meta-data support this would have to be custom developed to support capability description and exchange. JADE *Behaviours* are co-operatively scheduled [BCG07] and do not have any resource control nor any explicit support for evolution. The major drawbacks of this approach are that the capability model becomes JADE specific and the API (Application Programming Interface) only allows limited control over the adaptation process.
- *µCode:* µCode [Pic98] is a Java toolkit for code mobility. While it does not provide most of the features required, it does provide a flexible and lightweight approach to 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 in a *group* and also could be 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.

Since μCode does not place additional constraints on the type of classes that can be made mobile, it is possible to apply on top of it a standard for the capabilities. Other features required of a capability model have to be custom developed on top of μCode.

- *OSGi:* OSGi [OSG07] (originally the Open Services Gateway Initiative [MaK01]), 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, which can be further extended by developers [OSG03]. While heavier (in computational terms) than μCode, OSGi provides more features and is still lightweight since it can be deployed on small and embedded devices. In addition to a large population of available bundles [OSG07] (i.e. components), there are also multiple implementations of the OSGi framework (i.e. containers) available. Therefore, OSGi provides a suitable basis for building the VERSAG capability model.

We did not consider popular component models such as EJB (Enterprise Java Beans) and CORBA (Common Object Request Broker Architecture) [EmK02] which target enterprise applications because they are too unwieldy for use inside a mobile agent and also not suitable for the types of resource constrained environments targeted by VERSAG. The JADE `LoaderBehaviour` approach, while simple, is too restrictive and would make VERSAG agents JADE specific. μCode [Pic98] provides certain desirable features, but is not a standard. Furthermore, it is not actively maintained and is a minimal implementation, requiring considerable developments on top of it to meet our requirements. Thus, we decided to build the VERSAG capability model using the OSGi framework as a basis due to it providing many of the required

features, being an industry standard, being lightweight, and also having multiple compliant implementations available.

5.3.1 Performance Justifications for OSGi

The selection of OSGi does however raise the important issue of performance overheads. It is targeted towards a wide range of devices and application domains and has many features required by them, which can simply 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 [Ext99] was built. The solution consisted of creating two simple capability models, one OSGi based and one custom built, on top of JADE mobile 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 5-1.

Table 5-1: Comparison of custom-developed Vs OSGi based capabilities

Capability	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 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 conclude that the use of OSGi as the capability model implementation will not compromise our objective of building a lightweight agent framework.

Our next task is to select a suitable OSGi implementation.

5.3.2 Selection of an OSGi Implementation

We list below, five of the popular OSGi implementations currently available for use with VERSAG:

- *Apache Felix* (http://felix.apache.org/) is an open source implementation of the OSGi specifications.
- *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 is specifically targeted towards resource constrained devices [ReA07].

Of the above, we limited our selection process to the four open source implementations. Apache Felix, Knopflerfish and Equinox are all targeted for development of OSGi based applications on full powered computers (i.e. desktop and server computers) whereas Concierge has been designed from the beginning with resource constrained devices in mind. At less than 100KB, 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. However, 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 it is possible for us to later switch to a newer release if required.

5.3.3 Modifications to Concierge

Several 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 described next.

1. *Enable embedding Concierge within an application/agent*. Each VERSAG agent should have its own capability execution environment, which means that a VERSAG 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 Figure 5-2 shows how an OSGi container can be launched programmatically using this wrapper.

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

Figure 5-2: Launching Concierge programmatically

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. These are shown in Figure 5-3.

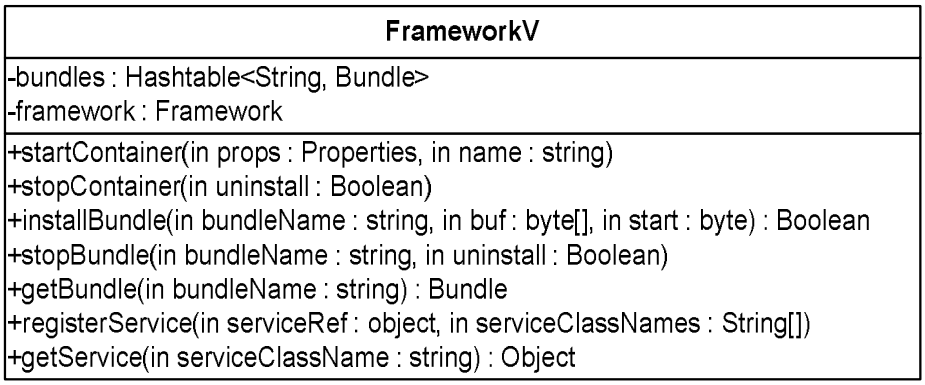


Figure 5-3: Wrapper class to enable embedding Concierge OSGi

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, with VERSAG we embed our OSGi container inside an agent that is mobile. Therefore, 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 passed into it at start up time.
3. *Disable caching of bundles and leaving 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, a mobile 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 disable the bundle caching implemented in Concierge through code modification.

These three modifications were incorporated into the most recent release of Concierge OSGi (version 1.0.0RC3). These modifications result in our container no longer being compliant with the OSGi specification. However, the container 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 100KB which is an important feature for VERSAG.

Having described the initial decisions on tool selection, we now present the design and implementation of our VERSAG agent framework.

5.4 VERSAG Prototype Design and Implementation

This section presents design and implementation details of the VERSAG agent prototype. The prototype agent design is based on the reference architecture presented in chapter 3 and is shown in Figure 5-4. The kernel, itinerary service, repository and capability execution service are implemented as key constituents of the agent. Three other services described in the reference architecture: capability exchange, adaptation and context, are implemented as capabilities in our design. As a consequence they become optional features of an agent. Therefore, an agent may decide not to carry an adaptation service when it is in a stable and resource rich environment. Similarly, an agent could support multiple capability exchange protocols by having different capabilities which implement different protocols. We describe each of the modules below.

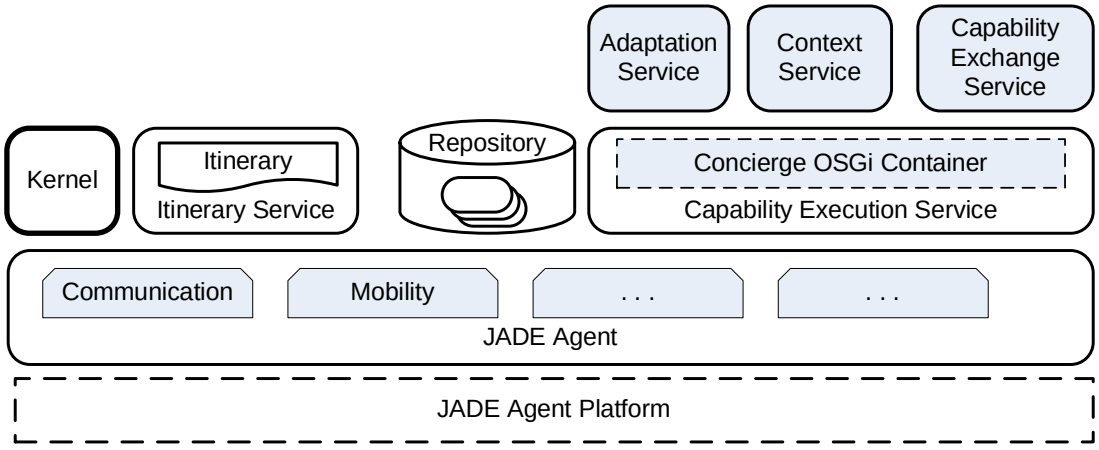


Figure 5-4: Architecture of prototype VERSAG agent

- *Platform specific agent:* The platform specific agent is JADE specific, and is realised as a class (`JadeAgent`) that inherits from the `jade.core.Agent` class. This is illustrated in the class diagram of Figure 5-5 (It should be noted that the diagram omits some relationships and secondary classes for purposes of clarity). Access to the agent's mobility service, platform directory service (i.e. Yellow Pages service) and agent messaging service are provided by this class. The agent contains a behaviour (`MessageListeningBehaviour`) which lets it listen for incoming agent messages. The `AgentIF` interface is used to provide a JADE/agent platform independent API for accessing the agent (i.e. access the repository, yellow pages, itinerary service or request mobility). This is useful when capabilities need to access these services but prefer to remain independent of the JADE platform.
- *Kernel:* The kernel, which is the main controller of a VERSAG agent, is a JADE `CyclicBehaviour` implemented by the class `KernelBehaviour`. The kernel is cooperatively scheduled on the single thread allocated to a JADE agent together with the `MessageListeningBehaviour`. The agent's `KernelBehaviour` gets its `action()` method invoked cyclically by the JADE behaviour scheduling mechanism. Once the behaviour gains control, an itinerary command is fetched from the `ItineraryService`. Figure 5-6 shows a sequence diagram illustrating two action cycles where the itinerary commands are to start a capability and then to stop it. In the first action cycle, the kernel retrieves an itinerary command and based on it instructs the

capability execution service to start execution of *capability A*. In the second action cycle, based on a *stop* command, the kernel instructs *capability A* to be terminated.

Java based mobile agent systems, as previously mentioned, provide weak mobility where agent state is not maintained across migrations. Thus, it is necessary for agents to explicitly save agent data when they migrate. The agent kernel therefore keeps track of its running capabilities so that they can be restarted after arriving at the new location. If migration fails due to any reason (e.g. unable to reach destination location, agent has non-mobile components), the agent reinitializes itself at the same location. The itinerant behaviour then identifies that the migration failed and passes control to appropriate error handling procedures.

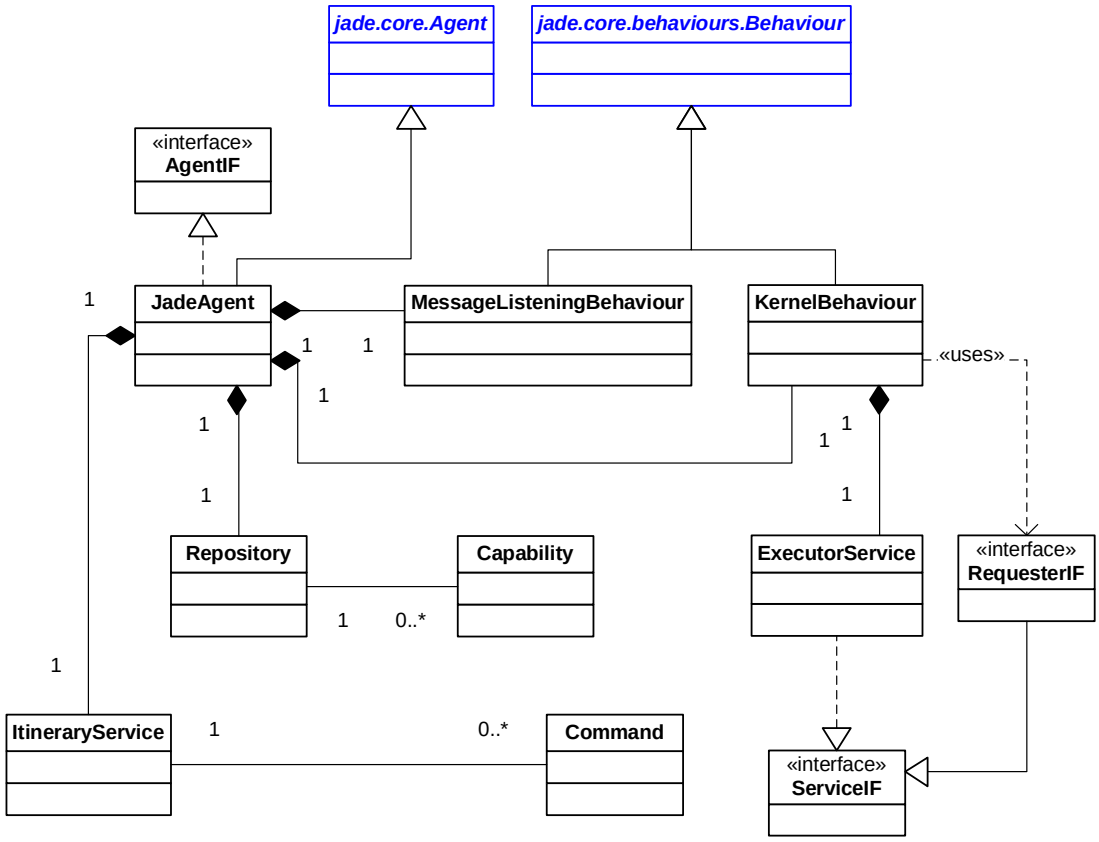


Figure 5-5: High-level class diagram of a VERSAG agent

- *Repository:* The repository, which is analogous to the capability repository of the reference architecture, stores capabilities as well as other agent

parameters which need to be maintained across agent migrations. It is created by the `JadeAgent` and needs to be accessible from within capabilities (i.e. OSGi bundles). Therefore, the kernel registers the repository as an OSGi service with the capability execution service (`ExecutorService`). Figure 5-7 illustrates the public operations available to use the `Repository`.

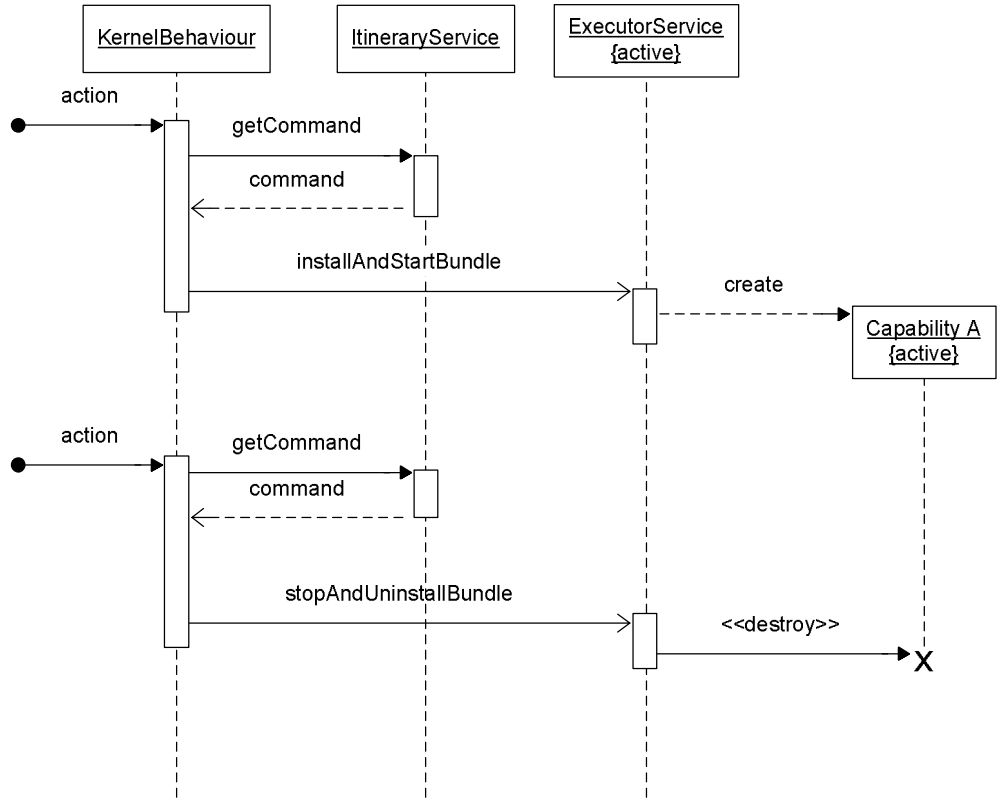


Figure 5-6: Sequence diagram of capability execution

- *Capability execution service:* This service is implemented by the `ExecutorService` class. The `ExecutorService` is launched and managed by the `KernelBehaviour` without any direct involvement of the `JadeAgent`. Upon agent start up, the service launches an embedded Concierge [ReA07] OSGi container on a separate execution thread. Thus, VERSAG agents are multi-threaded unlike normal JADE agents. This service has to be shut down before agent migration (or termination) and needs to be started after an agent's initial creation and arrival at a new location. The API exposed by the service is shown in Figure 5-8. The kernel makes use of this API to start and stop capabilities. Objects outside the OSGi container (e.g. the

Repository) are registered as services so that they are accessible to capabilities executing within the OSGi container. Capabilities which are registered as services (e.g. the Capability Exchange Service) can be accessed outside of the OSGi container using the `getService()` method.

- *Itinerary service:* This service is implemented by the `ItineraryService` class. It provides methods to store an itinerary and to update and retrieve itinerary commands. The sequence diagram of Figure 5-6 shows how itinerary commands are fetched by the agent kernel.
- *Capability exchange service:* This service, as earlier mentioned is optional and is implemented as a capability. The interface `RequesterIF` needs to be implemented as part of this module to represent the capability search and acquisition side.

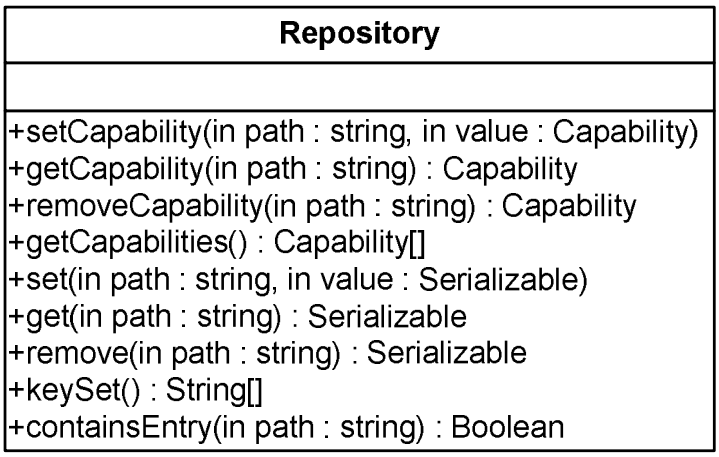


Figure 5-7: Repository interface

We do not delve into implementation details of the context service module as it is not the focus of this research. Details of an adaptation service capability are provided later in this chapter.

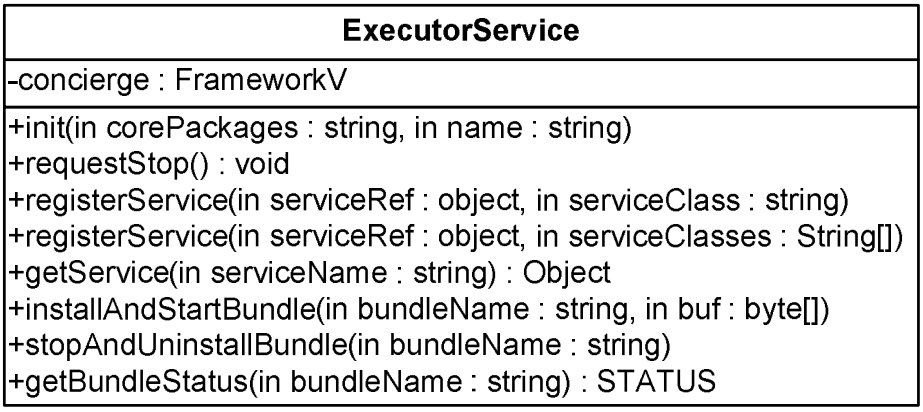


Figure 5-8: Capability Execution Service interface

This section so far described the internal architecture and design details of a JADE based VERSAG agent. It is expected that VERSAG capabilities would be developed by third-parties and be reusable across multiple agents and different applications. Therefore, in the next sub-section we describe the various interfaces that have to be adhered to in developing capabilities.

5.4.1 Interfaces for Capability Development

One of the requirements of a VERSAG capability model that was identified in chapter 3 is a set of well-defined and minimal interfaces according to which capabilities can be built. Here we use the term interface to refer to programmatic interfaces (e.g. Java interfaces [AGH05]) as well as other restrictions such as required file formats and naming schemes. This sub-section looks at this important aspect.

VERSAG by default does not impose any interface restrictions on a capability other than those required by the OSGi specifications. Therefore, any OSGi release 3 (R3) [OSG03] compliant bundle supplemented with the meta-data required by VERSAG (described later in this sub-section) can be used as a capability. If a capability has to interact with VERSAG agent features, then further interface restrictions apply. Furthermore, capabilities can define their own interfaces in order to enable interactions between capabilities. Figure 5-9 illustrates these three levels of interfaces applicable to capabilities. We describe them below.

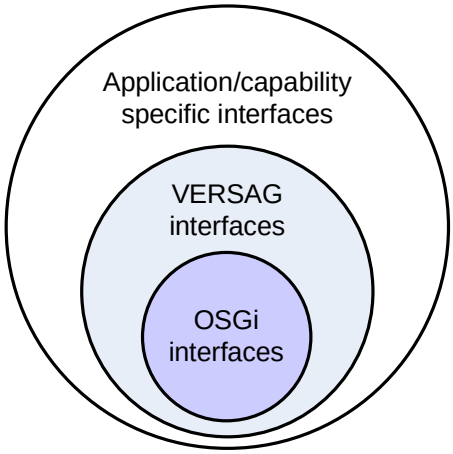


Figure 5-9: Types of interfaces for VERSAG capabilities

5.4.1.1 OSGi Specific Interfaces

The capability model used in our prototype is, as previously described, based on the OSGi framework and a capability is implemented in the form of an OSGi bundle. 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 manifest file consists of header-value pairs as shown in Figure 5-10 below.

```
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"
```

Figure 5-10: Capability bundle manifest

Of the headers shown, the `Manifest-Version` header is mandatory for all JAR files while the others shown are specific to OSGi. Furthermore, only the `Bundle-SymbolicName` header which assigns a unique name to a bundle is mandatory. Since OSGi is required to ignore any unknown headers, it is possible for VERSAG to define new headers and still have the capabilities be executable in standard OSGi containers. The manifest specifies a `Bundle-Activator` header which points to a

special bundle activator class that should be found in the JAR file and implement the `org.osgi.framework.BundleActivator` interface. This interface specifies `start()` and `stop()` methods which are invoked by the container to start and stop the bundle respectively. The code excerpt in Figure 5-11 shows a Bundle Activator class with these two methods.

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

public class ABCActivator implements BundleActivator  {

  public void start(BundleContext context) throws Exception {
      //initialize and start capability
      ...
  }

  public void stop(BundleContext context) throws Exception {
      //stop capability and do any clean up
      ...
  }
}
```

Figure 5-11: Capability bundle activator

An Activator class needs to be declared `public` and also have a default constructor (i.e. constructor with no arguments). A bundle which only exports its classes for use by others and does not have any active component (i.e. a *passive* capability) may omit the bundle Activator class. Such a bundle would use the `Export-Package` header to specify what classes it is making available for other bundles to use. It is necessary to group classes to be exported in (Java) packages as the header takes package names rather than individual class names. Similarly, bundles which depend on classes exported by other bundles can inform the container of this using the `Import-Package` header. This enables management of dependencies between bundles. The `Bundle-RequiredExecutionEnvironment` manifest header indicates the execution environments that the bundle requires for its execution. Execution environments are defined by the OSGi Alliance[10] and define a minimum set of APIs available in it which can be used by the bundle.

[10]http://www.osgi.org

The `BundleContext` object that a capability receives (via *start*/*stop* methods) is the pathway to communicate with the OSGi container. It is through the `BundleContext` that a capability gets access to its owning agent and other services registered in the OSGi container. The code excerpt in Figure 5-12 illustrates how references to the `AgentIF` and an imported service named `DataSource` are obtained by a capability.

```
AgentIF agent;
DataSource source;
BundleContext context;
ServiceReference ref;
...
ref = context.getServiceReference(
              AgentIF.class.getName());
agent = (AgentIF) context.getService(ref);
ref = context.getServiceReference(
              DataSource.class.getName());
source = (DataSource) context.getService(ref);
...
```

Figure 5-12: Programmatically obtaining a reference to AgentIF

The OSGi specifications [OSG03] provide further details of headers, bundle life cycle, OSGi services, managing classes of separate bundles etc.

5.4.1.2 VERSAG Specific Interfaces

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.itinerary.ItineraryService`
- `mma2.bundles.common.AbstractOneShotClass`

These interfaces are required for tasks such as making use of agent services (`AgentIF`), retrieve/update data stored in the agent's state (`Repository`) and modify the agent itinerary (`ItineraryService`). Capabilities such as the *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. Application-specific capabilities (e.g. a capability which implements a data mining algorithm) are less likely to need to use these interfaces.

As discussed in chapter 3, a capability is described with a unique identifier, function descriptions, list of supported platforms, execution type and various meta-data. In the current prototype, these have to be specified as name-value pairs in a text file which is read by the agent at start up and associated with capabilities.

5.4.1.3 Application Specific Interfaces

VERSAG does not provide any guidelines on the granularity or functions implemented per capability. However, it is beneficial to build capabilities in a way that promotes reuse. In order to facilitate this, we recommend a service-oriented approach with capabilities classified into two types as follows:

1. *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.
2. *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, the personal assistant agent Bob from our running example would use such a capability together with different types of "document reader" capabilities in order to search the documents in the office

Intranet. 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.

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 here that it is possible to have monolithic capabilities which combine these two types of functionality. However, they do not promote reusability and therefore should be limited for use in special circumstances.

Capabilities are a key concept of VERSAG, and in the next two sub-sections we describe two sets of capabilities developed within our VERSAG prototype implementation. They also illustrate how service and coordinating types of capabilities use application specific interfaces to interact.

5.4.2 Adaptation Cost Model Capability

In chapter 4 we proposed a multi-criteria decision making cost model which allows VERSAG agents to make cost-efficient capability selection decisions. In this sub-section we describe the implementation of this cost model within our VERSAG prototype.

Cost model invocation for capability acquisition in our implementation involves two itinerary operations.

1. The first itinerary operation invokes the capability exchange service to search peer agents for capabilities that match the requirements and retrieve their responses (containing details of matching capabilities).
2. The second operation invokes the cost model to apply the multi-criteria selection procedure and choose the most cost-efficient capabilities from

amongst those available. Relevant cost components are specified as part of the itinerary command. The operation continues to acquire the selected capabilities and completes by updating the agent itinerary.

Following the service-oriented approach presented in the previous sub-section we have implemented the cost model as a combination of three capabilities. These three capabilities are named *AhpCostModel*, *IRequester* and *ISelector*, and the component diagram of Figure 5-13 illustrates the dependencies amongst them. A fourth capability of PASSIVE execution type used to define the interfaces and common classes required by the adaptation cost model is not shown in the diagram for the sake of clarity. These interfaces are not imposed by the agent framework and are application specific as defined in the previous sub-section. All capabilities however do need to adhere to the OSGi specific interface. This is indicated in Figure 5-13 by having them provide an `OSGiIF` interface which the Concierge OSGi component requires.

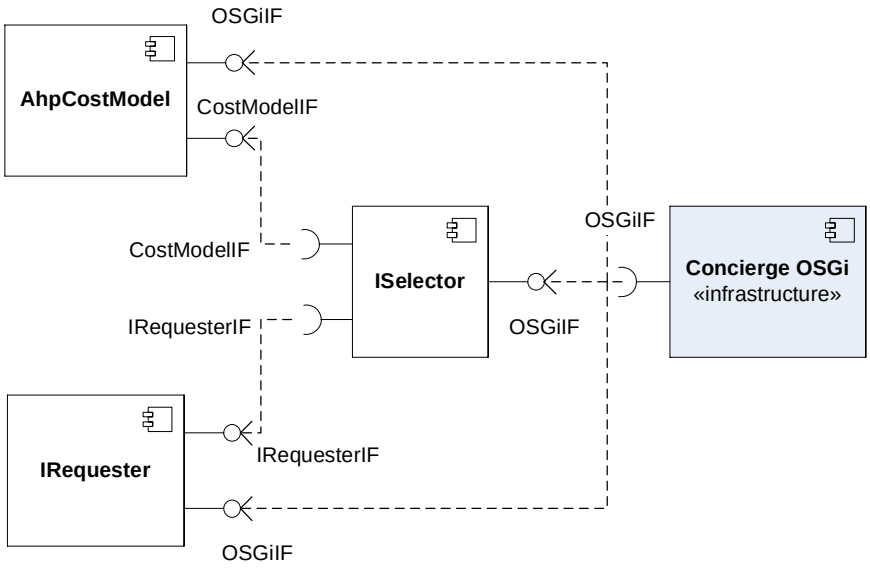


Figure 5-13: Component diagram of adaptation cost model

We present brief descriptions of the three capabilities next:

- *IRequester:* The purpose of this capability is to provide the cost model with the ability to request and acquire capabilities from peer agents. For this, it implements the `IRequesterIF` interface which is part of the capability

exchange service. The *IRequester* capability therefore also belongs to the capability exchange service. The protocol used for capability exchange is encapsulated within the capability and it is expected that different capabilities using different protocols would be implemented by third parties. The *IRequester* is therefore a service capability.

- *ISelector:* This is the coordinating capability which brings together the other two capabilities to achieve the required goals. Its responsibilities include identifying possible sets of capabilities (i.e. alternatives/capability groups) based on search responses from peer agents, building the constraints that apply to the current activity, invoking a cost model capability to sort the alternatives, acquiring selected capabilities and finally updating the itinerary to use the acquired capability instances. It needs a capability implementing the `CostModelIF` interface to sort the alternatives and a capability implementing the `IRequesterIF` interface in order to retrieve capabilities. This is illustrated in the component diagram as required interfaces. Furthermore, *ISelector* needs access to the VERSAG specific interfaces (`AgentIF` and `ItineraryService`) for itinerary updating (this is not shown in the diagram).
- *AhpCostModel:* This is another service capability built according to the `CostModelIF` interface to accept a set of possible alternatives (i.e. capability groups) and cost constraints, and return the least cost alternative. This function is generic enough to be applicable to different cost based selection situations. Thus, it is expected that third-party developers would create capabilities which fulfil this interface. The *AhpCostModel* capability is one such, which uses the Analytic Hierarchy Process (AHP) for its internal decision making.

5.4.3 Information Extraction Capability

We now describe a set of capabilities which provide an agent with the ability to read different types of data sources and extract information from them. This information extraction capability set is used in the experiments described in chapter 6.

Specifically, the personal assistant agent Bob from our running example makes use of these to search documents in the office Intranet.

Figure 5-14 shows the involved capabilities and their dependencies. Dependence on the Concierge OSGi container is omitted to improve clarity. The information retrieval process requires three types of capabilities which we describe next:

- *File Processor:* This is the coordinating capability. It 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 data source and invoked. The *File Processor* then stores any extracted information in the agent's *Repository*.
- *Result Handler:* This capability reads information stored in the Repository and displays them. It does not have any dependency or contact with the capabilities which extract and store information (i.e. the *File Processor*). It only needs to be aware of the key under which the information is stored and the format of information. Furthermore, how the information is "handled" depends on application requirements. It could be to simply display them (as we illustrate in chapter 6), to further process, or to transfer to another agent. This capability is therefore a service capability.
- *Reader:* Figure 5-14 shows three capabilities which fall under this category: one to read text files (*TxtReader*) and two implementations for PDF files (*jPodPDFReader, PdfBoxReader*) from different vendors. They implement the `ReaderIF` interface and contain the generic function of reading contents from a given data source (i.e. a text or PDF file in this case). This is a useful functionality in many situations, and therefore these are service capabilities. While the capabilities we have implemented read different types of data files, the `ReaderIF` interface can be used to read data from other sources such as database tables and network sockets.

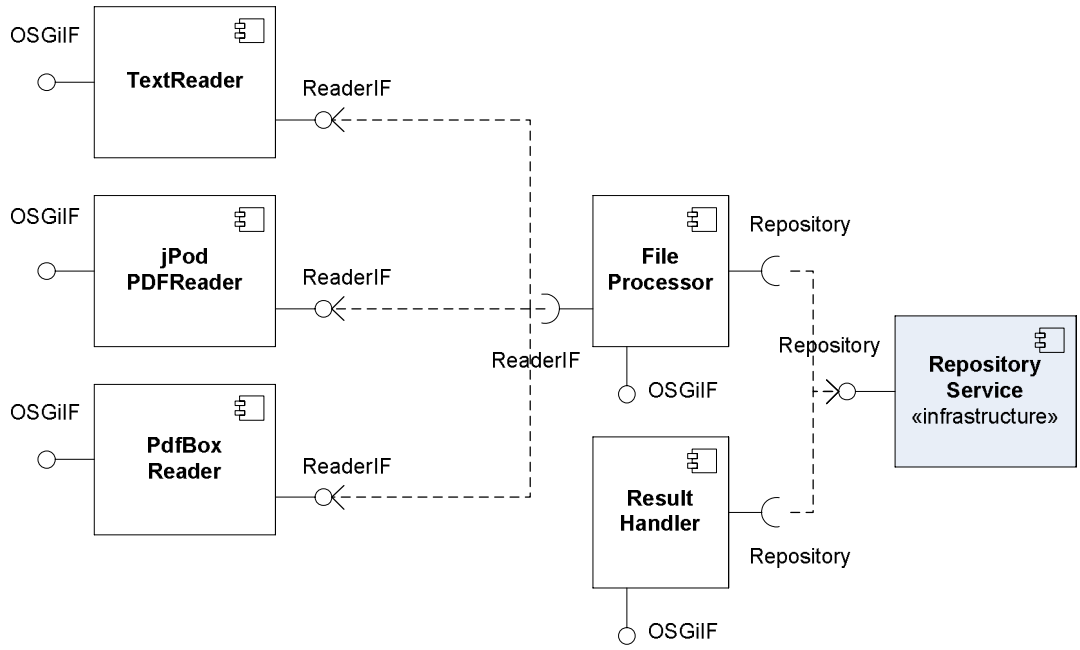


Figure 5-14: Component diagram for an information retrieval capability set

In this section we described design and implementation details of the VERSAG framework prototype. We now proceed to describe the deployment and operation of our VERSAG prototype implementation.

5.5 Operation of the VERSAG Prototype

This section presents an overview of three aspects related to the operation of the VERSAG prototypical implementation. We start with a description of the distribution format of the prototype and then describe tools for monitoring and managing VERSAG agents. Finally, we present the options available to start and control VERSAG agents.

5.5.1 VERSAG Prototype Distribution

In this sub-section we present details of the VERSAG distribution. Our prototype implementation, being a Java based system, is distributed as a collection of Java archive (JAR) files. These JAR files need to be made available to the JADE runtime (i.e. needs to be in the Java class path of the JADE runtime) in order to create VERSAG agents and management tools. These archives are as follows:

- *Core libraries:* The core agent framework consists of the agent class for creating VERSAG agents (i.e. `mma2.jade.JadeAgent`) and associated classes and interfaces. These are available in a JAR file named *versag1.jar*.
- *Tools library:* The two tools developed for managing and monitoring the execution of a VERSAG based multi-agent platform are distributed in a JAR file named *tools.jar*. We describe these tools (Controller Agent and the Remote Monitoring Console) later in this section.
- *Capability libraries:* Each VERSAG capability is contained in a separate JAR file of its own. Locations of these files and their associated meta-data (i.e. capability descriptions) are stored in a configuration file which is read by agents at agent creation time.
- *OSGi implementation:* In addition to the above, since the Concierge OSGi container is used inside our agents, it is necessary for the modified Concierge class libraries located in *concierge-1.0.0.RC3.jar* to be available to the JADE runtime where agents are created.

Appendix B shows the folder structure at the main container for our prototype.

5.5.2 Monitoring and Management Tools

We now proceed to a description 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). We first present a brief overview of the RMA and then describe two tools developed specifically for VERSAG: the *Controller Agent* (CA) and the *Remote Monitoring Console* (RMC).

JADE Remote Monitoring Agent (RMA)

JADE provides a graphical management interface named the RMA (Remote Monitoring Agent) in the form of a mobile agent. The RMA shows the agent platform, its containers (locations) and agents executing at each container in a tree structure as shown in Figure 5-15. Any other FIPA [Fip10] compliant agent platforms that it is aware of can also be displayed.

It allows creating agents (Figure 5-16), deleting agents, 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 GUI. 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 [BCT07].

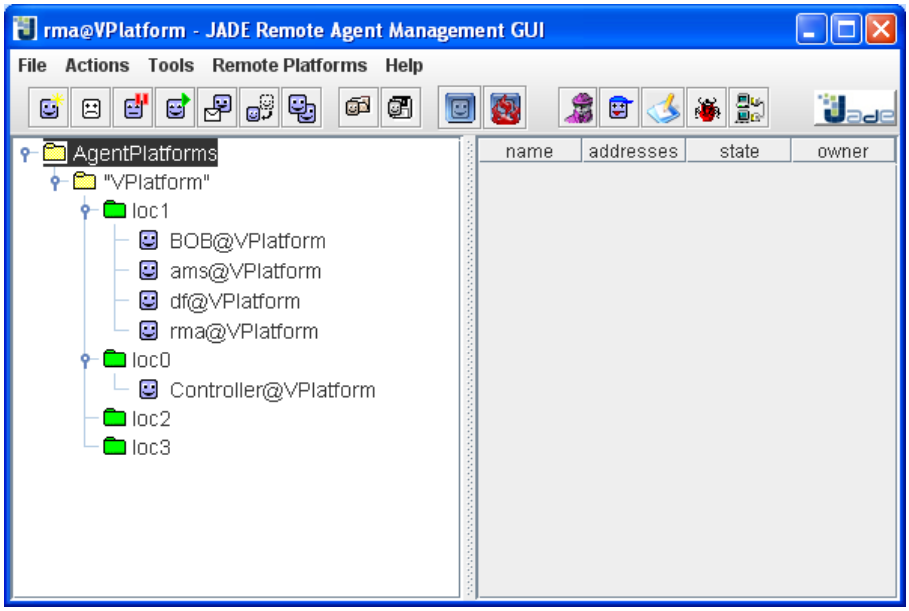


Figure 5-15: JADE Remote Monitoring Agent (RMA)

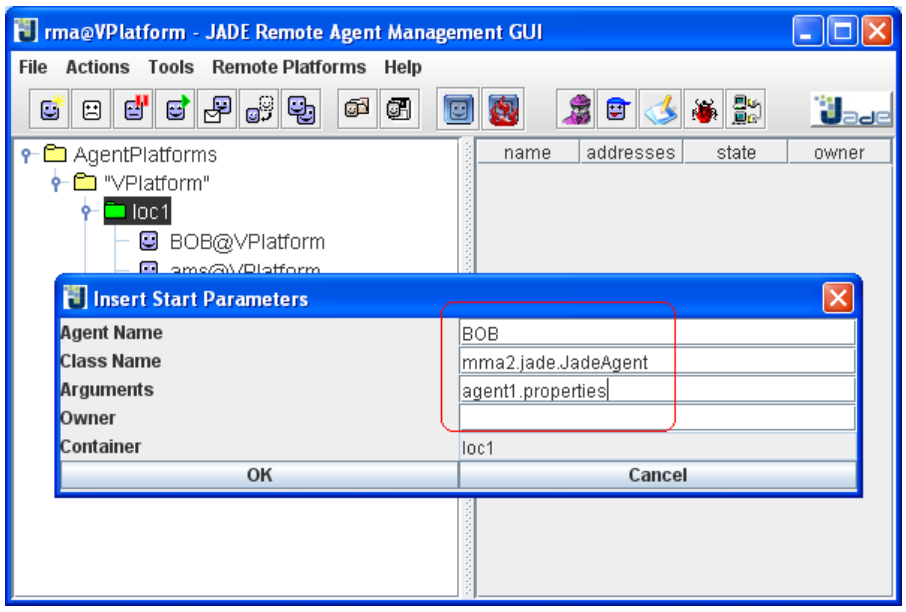


Figure 5-16: Creating a new agent using the JADE RMA

Controller Agent (CA)

The Controller Agent (CA) is a special agent developed to enable administration of VERSAG agents through a command line interface. It is not a VERSAG agent.

While the RMA agent and associated JADE tools allow sending messages to agents via graphical interfaces, they can be cumbersome when a large number of messages have to be sent. Furthermore, they cannot be used when a Windowing system is not available (e.g. on headless Linux machines and when remotely logging in via SSH protocol). The CA provides a more convenient command-line and script file based approach to control agents. With the CA, agents can be controlled using simple text based commands or multiple commands can be encapsulated in a script file that can then be executed through the CA's interface. The script file commands currently supported are:

- Create a new agent
- Send a message to an agent
- Move an agent (via an itinerary request)
- Kill an agent
- Kill a JADE container (i.e. a location)
- Shut down the agent platform
- Request an agent's status
- Pause script execution for a specified amount of seconds
- Quit the Controller Agent

It is possible also to run any of these commands, except sending messages, from the CA's command line. An excerpt from one of the CA script files used in our evaluations (chapter 6), and a linux script to start a JADE container with the CA is given in Appendix B. A screen capture of the running CA listing its command syntax is shown in Figure 5-17. The response received by a status request through the CA is shown in Figure 5-18. A status response contains the agent's name, location, internal parameters and itinerary.

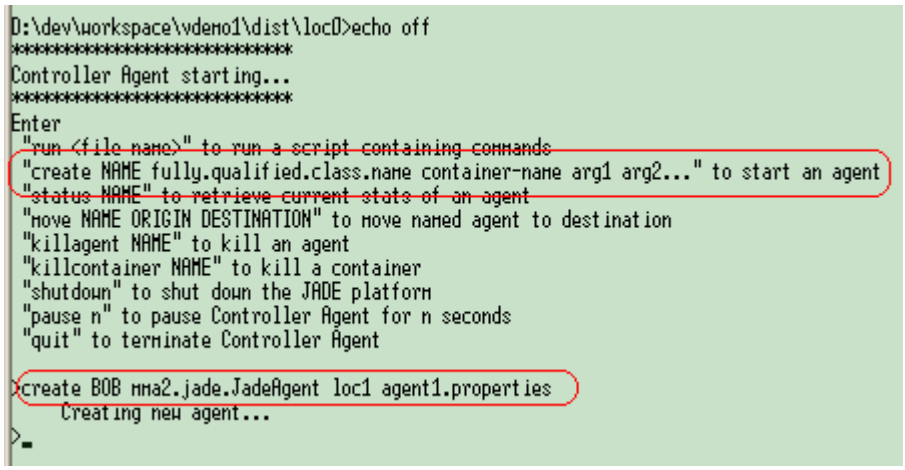


Figure 5-17: The Controller Agent (CA) interface

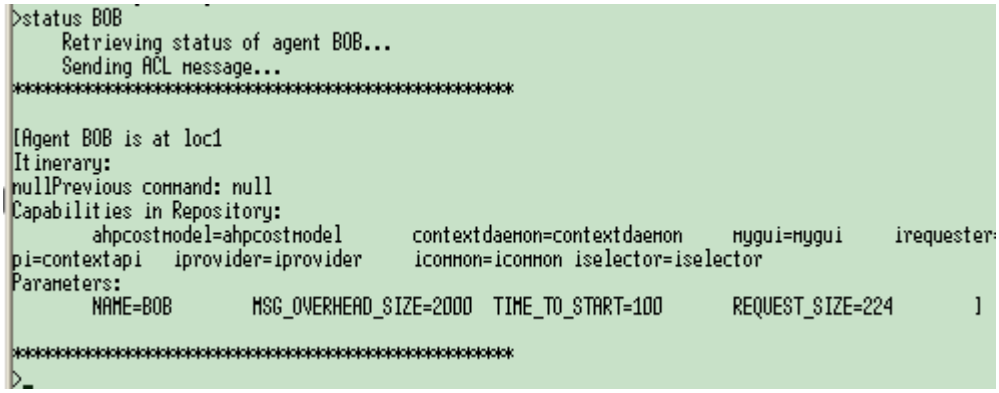


Figure 5-18: Controller Agent shows an agent's status

Remote Monitoring Console (RMC)

The Remote Monitoring Console (RMC) is a tool which displays specially formatted messages to provide a user with a bird's eye view of the functioning of a VERSAG multi-agent system. It is a graphical tool executed outside the agent system and listens over the network for messages coming from JADE containers. These are then displayed in the received sequence via a graphical interface as shown in Figure 5-19. VERSAG agents as well as capabilities use the RMC to inform users of their activities by incorporating appropriate log statements.

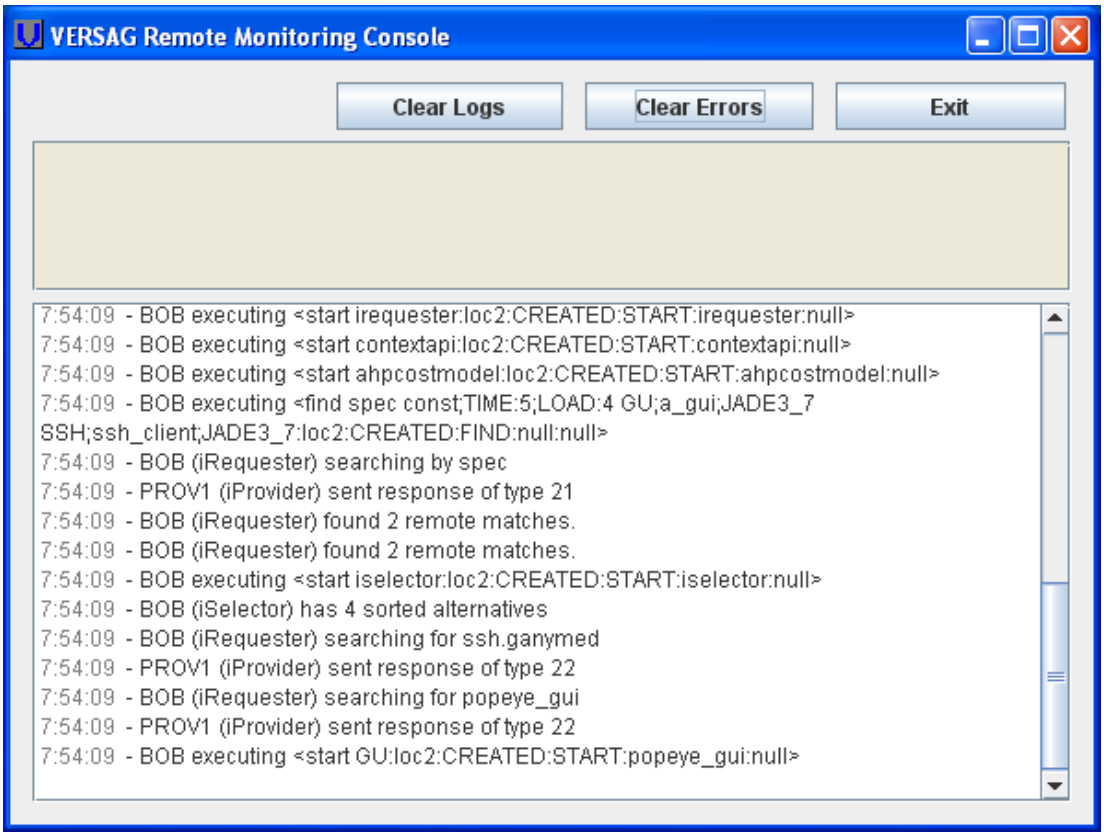


Figure 5-19: VERSAG Remote Monitoring Console

Our agents and capabilities produce detailed log messages to describe their actions and to highlight any exceptional situations faced. Log statements are typically written to a text file named *jade.log* at each JADE container. Appendix B contains log statements recorded from one of our evaluations. The RMC makes use of the logging mechanism to receive messages from agents. However, since each JADE container creates a network connection to the RMC over which log messages are sent, the use of the RMC results in extra overheads in terms of processing and network load. Appendix B contains the command used to start the RMC.

5.5.3 Running VERSAG

We now present the options available for creating and controlling VERSAG agents. As mentioned previously, when the JADE runtime is started, the VERSAG specific libraries have to be added to the Java classpath of the container where VERSAG agents are to be started.

Creating Agents

A new agent can be created either using command line options when starting the JADE container, through the RMA agent or via the Controller Agent. Figure 5-16 illustrates how to use the RMA GUI to start a new agent. Figure 5-17 illustrates how to use the Controller Agent to start a new agent. To create a VERSAG agent, the following parameters have to be specified:

- A unique name for the agent (e.g. "BOB")
- Fully qualified class name of the agent (`mma2.jade.JadeAgent` for VERSAG)
- Location/container in which the agent is to be started (not required if specified when starting the container)
- A single argument specifying the configuration file with agent start up details

The configuration file is our approach to populate an agent with capabilities and other relevant data at start up.

Controlling Agents

The primary mechanism to control VERSAG agents is through agent messages in the Agent Communication Language (ACL). The agent treats any incoming message with the REQUEST performative (of the FIPA-ACL message structure [BCG07]) as a command message. The supported commands are as follows:

- Request an agent for its status (details of itself)
- Update the agent's internal parameters
- Supply a new itinerary to the agent
- Request an agent to *terminate* itself
- Request an agent to *move* to a specified location
- Request an agent to *clone* itself

While moving and cloning of an agent is supported via these commands, it is preferred to request agent migration through itinerary requests. Cloning is not used in VERSAG. The RMA GUI is the JADE provided mechanism to send messages to

agents. It is also possible to send messages to agents through Controller Agent script files. This is the approach we use in our experimental evaluations, described in the next chapter. Appendix B presents commands and sample scripts used for the operation of the VERSAG prototype.

5.6 Summary

In this chapter we described our VERSAG agent framework prototype implementation. The guiding criteria behind the VERSAG platform development and tool selection were presented first, followed by details of capability model implementation selection and associated issues. We provided a detailed description of the agent framework implementation with the aid of UML (Unified Modeling Language) diagrams. We also described the types of interfaces that need to be adhered to when developing capabilities. Two sets of capabilities were described in order to illustrate their usage. Finally, the chapter described operation of the prototype, detailing how to start and run VERSAG and the tools available for monitoring and managing agents.

The prototype implementation described in this chapter was carried out with the aim of providing a basis for experimental validation of the ideas proposed in chapters 3 and 4. These include demonstrating the feasibility of mobile agents with the feature of dynamic capability exchange, and evaluating the effectiveness of the cost model for improving performance of VERSAG agents in pervasive computing applications. In the next chapter we describe the experiments carried out towards this end and analyse their results.

C H A P T E R S I X

6 VERSAG Evaluation

The previous chapter presented implementation details of our VERSAG prototype. The prototype was built with the aim of providing a basis for experimental validation of concepts proposed in earlier chapters. In this chapter we describe experiments carried out and analyse the results obtained.

6.1 Introduction

We have proposed in this thesis that mobile software agents which can adapt compositionally at runtime can be an effective enabling technology for building pervasive applications and services. Chapter 3 formally defined the VERSAG agent framework which encapsulates this proposal. We defined an agent as a lightweight itinerary driven mobile entity which gains new behaviours at runtime by acquiring reusable software components termed *capabilities*. These capabilities can also be discarded when no longer needed. We further developed a model that enables agents to share their capabilities with peer agents instead of depending on a centralized capability source. This is beneficial in the uncertain environments encountered by pervasive applications. In chapter 4, we proposed a multi-criteria cost model to assist agents in making cost-efficient adaptation decisions. Time, network load, computational resource needs and accuracy of output were the key cost criteria considered in our model. The chapter also presented approaches for estimating these cost components.

In this chapter we use our implementation prototype presented in chapter 5 to experimentally evaluate and demonstrate the benefits of VERSAG. Within the context of the concepts proposed in preceding chapters, we identify the following evaluation criteria which need to be addressed:

1. *Feasibility and functional benefits of component sharing mobile agents.* The concept of sharing functional components among peer agents at runtime is a key distinguishing feature of VERSAG agents. Therefore, it is necessary to demonstrate the technical feasibility of this concept and that such agents bring about functional benefits not achievable with conventional mobile agents.
2. *Performance benefits of component sharing mobile agents*. We have proposed that component sharing mobile agents can lead to improved performance for pervasive computing scenarios in general, and improved migration efficiency in particular. Therefore, we need to validate whether VERSAG agents can bring about performance benefits as stated.
3. *Performance of decision making cost model*. The multi-criteria decision making cost model was proposed to assist agents make cost-efficient adaptation decisions. It is vital therefore to evaluate that the cost model is capable of making accurate adaptation decisions that lead to overall performance improvements.
4. *Scalability of component sharing mobile agents.* The usefulness of VERSAG is maximized when there is a large distributed ecosystem of agents engaged in diverse activities and interacting with each other. Therefore, it is necessary to evaluate how VERSAG agents perform when deployed in a large multi-agent system and need to interact with large numbers of agents.

By performance, we refer specifically to network load and time consumption. The experimental evaluations described in this chapter were designed with the aim of addressing the above evaluation criteria. Since our intention was not to develop sophisticated and complex capabilities, the capabilities used in these experiments tend to implement simple application-level functions. However, we note that the lack of complex capabilities has no notable impact for our evaluations. For each

experiment, we describe the specific objectives, experimental setting, results, and present an analysis of results.

The rest of this chapter is organized as follows. First, in section 6.2 we present a case study implementation to investigate the feasibility and functional benefits of VERSAG. Then, in section 6.3 we describe two experiments carried out in order to investigate VERSAG's performance benefits. These experiments investigate agent migration efficiency and benefits of localized capability sharing. The adaptation cost model proposed for capability acquisition decision making is evaluated in section 6.4. Section 6.5 is devoted to describe experiments investigating the scalability of VERSAG. Finally, in section 6.6 we conclude the chapter.

6.2 Evaluating Feasibility and Functional Benefits

A fundamental advantage VERSAG agents have over conventional mobile agents is the ability to acquire new behaviours for situations which are not anticipated at design time. Thus, a VERSAG agent is able to continue execution in situations where a conventional mobile agent would have to be replaced with a new one. It is this ability which makes them especially suitable for use in dynamic pervasive environments. In this section, we describe a case study scenario in order to demonstrate and establish the feasibility and functional benefits of VERSAG agents. The criteria validated are given below:

1. The ability of an agent to dynamically shed capabilities that it contains (in order to be lightweight when migrating over a wireless link).
2. The ability of an agent to dynamically acquire new capabilities and use them to fulfil an allocated itinerary.
3. The ability of an agent to search for and acquire capabilities it needs from other agents.
4. The ability of an agent to make a cost-efficient selection based on contextual input and user preferences when multiple suitable capabilities are available from others.

Experimental Setting

The case study is based on the scenario introduced in chapter 3 which describes a virtual personal assistant agent helping a human user (Bob) automate some of his office work on a typical workday morning. We now describe a slightly modified version of the scenario which we use for evaluation. While on his way to work by train, Bob types a memo on his mobile computing device, which needs to be uploaded to the office virtual noticeboard. Once he arrives at his 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 their m-commerce product *mzone kiosk*. Bob indicates that he would like to access this list of documents from his laptop computer.

In our experiment, 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 devices such as mobile phones and personal digital assistants (PDA)[11]. 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[12] page named "NoticeBoard". XWiki was chosen since it allows REST style [RiR07] 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 [BCG07]) for the experiment. Table 6-1 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 Figure 6-1.

[11] As discussed in chapter 1, this is primarily due to limitations of software development environments such as lack of dynamic class loading on mobile Java versions.
[12] http://www.xwiki.org

Table 6-1: Locations of the case study agent environment

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

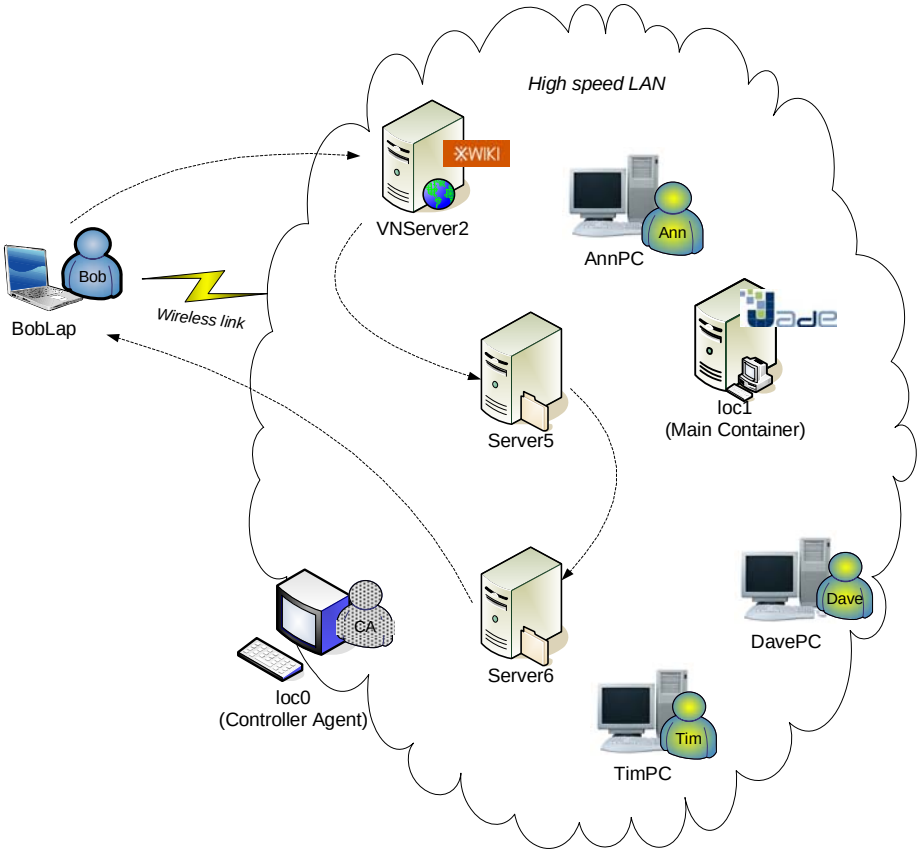


Figure 6-1: Overview of agent environment for case study

The JADE Remote Monitoring Agent (RMA agent) also presents an overview of the agent distribution on the platform, as seen in Figure 6-2.

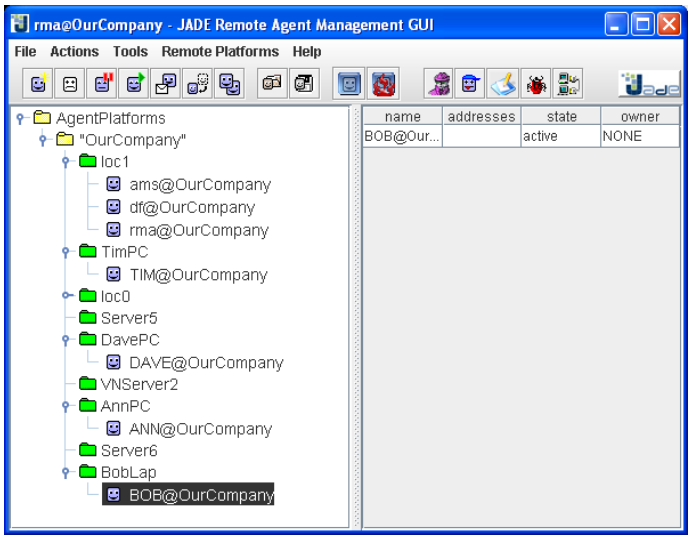


Figure 6-2: RMA agent showing agent distribution on the platform

Agents Dave, Ann and Tim are the "peer" agents with whom agent Bob may share capabilities. The capabilities contained in these agents at the start of the experiment are shown in Table 6-2.

Table 6-2: Capabilities carried by agents at the start of experiment

Agent	**Capabilities in agent's repository**
Dave	icommon, iprovider, contextdaemon pagui httpclient, xwiki, vnclient, editor 2common, 2matcher, 2result, resultgui 2txtreader, 2jpodpdfreader, 2pdfboxreader insertion, shell, idler, random
Ann	icommon, iprovider, contextdaemon pagui httpclient 2result, resultgui, 2txtreader idler
Tim	icommon, iprovider, contextdaemon gui, acme_gui, popeye_gui ssh.ganymed, ssh.duplicate, imapigtalk, imui, imapi, txt2pdflocal, txt2pdfws
Bob	icommon, irequestor, contextapi iselector, ahpcostmodel pagui editor

The purpose of each capability relevant to the current experiment is briefly outlined in Table 6-3. Capabilities not required for our experiment are not described.

Table 6-3: Description of capabilities

Capabilities	**Description**
icommon	Common classes needed for capability search and request
iprovider	Server capability which listens for incoming capability requests and responds to them
irequester	Client capability which lets agent request capabilities from other agents
iselector	Coordinating capability for capability acquisition decision making
ahpcostmodel	Implements a multi-criteria decision making process based on the Analytic Hierarchy Process
contextdaemon	Context server capability used to measure network parameters
contextapi	Context client capability to measure network parameters
pagui	Agent GUI to show its current status
editor	A simple memo typing capability
httpclient	Passive capability which implements HTTP methods
xwiki	An XWiki client capability
vnclient	A coordinating capability for the Virtual noticeboard client
2common	Common classes needed for information retrieval/search
2matcher	Coordinating capability for information retrieval/search
2txtreader	Text file reader for use in information retrieval
2jpodpdfreader	PDF file reader for use in information retrieval. Uses *JPod*[13] library
2pdfboxreader	PDF file reader for use in information retrieval. Uses *PdfBox*[14] library
2result	Console based display for aggregated search results
resultgui	GUI to display aggregated search results

Description of Experiment

In this sub-section we describe the experimental steps carried out to illustrate the feasibility and functional benefits of VERSAG agents.

[13] http://sourceforge.net/projects/jpodlib/
[14] http://pdfbox.apache.org/

Step 1: Demonstrate agent migration

We demonstrate the ability of the VERSAG agent Bob to migrate from the laptop (BobLap) to a device on the LAN (Server5). This is primarily an illustration of the fact that the migration capability of an agent remains intact. Bob's GUI is started on the laptop to verify the capabilities it contains. The screen capture on the left-hand side of Figure 6-3 shows the agent GUI where we observe that the agent is located at "BobLap" and that it contains the seven capabilities as listed in Table 6-2. The bottom text area shows the most recent itinerary command issued to the agent, which is to start up the GUI (i.e. *pagui* capability).

Agent Bob is instructed (i.e. issued an itinerary through the Controller Agent interface) to move to *Server5* and the data transferred between the two devices is measured using the network protocol analyser wireshark [LSW11]. Log messages received by the Remote Monitoring Console (RMC) and the RMA GUI indicate that Bob successfully migrated. When the agent is instructed to start the GUI, it starts up on the *Server5* computer as shown in the screen capture on the right-hand side of Figure 6-3, thus confirming that agent Bob has the ability to migrate.

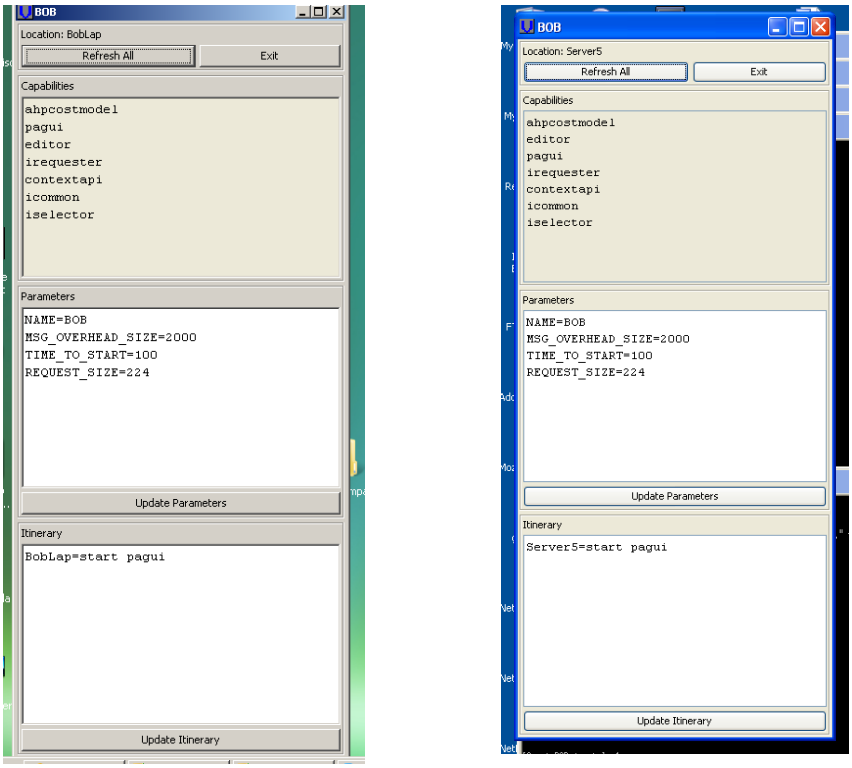


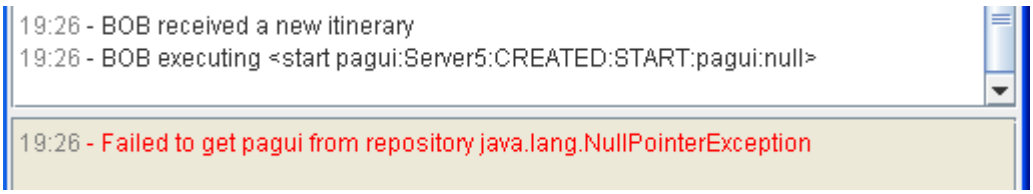
Figure 6-3: Agent Bob's starting status as seen on the laptop and Server5

The agent is again instructed to return to the laptop and data transferred is measured. IP level data traffic observed for the two transfers are shown in Table 6-4.

Table 6-4: Summary of IP data traffic as agent Bob migrates

Agent Travel Direction	Total		A to B		B to A	
	Packets	Bytes	Packets	Bytes	Packets	Bytes
Laptop (B) to Server5 (A)	148	88470	57	3789	91	84681
Server5 (A) to Laptop (B)	112	85168	73	82718	39	2450

It can be seen that on average 82KB of data is transferred in the direction of agent migration. Next, we repeat this process but request the agent to discard specific components namely, the *editor*, *ahpcostmodel* and *pagui* capabilities before migration. Starting the agent GUI at the destination (Server5) fails with error messages as shown in Figure 6-4 (RMC error messages and corresponding error in log file). The error message confirms that the *pagui* capability (which the agent discarded) is no longer in the agent's repository. We note that this is merely to illustrate the operation of discarding components.



24/07/11 19:26 [12] INFO mma2.jade.KernelBehaviour - BOB executing <start pagui:Server5:CREATED:START:pagui:null>
24/07/11 19:26 [12] SEVERE mma2.jade.KernelBehaviour - Failed to get pagui from repository java.lang.NullPointerException

Figure 6-4: Errors when starting Agent GUI (RMC and log messages)

IP level data traffic for migration after discarding the three capabilities is shown in Table 6-5. We note that the mean difference in data traffic in the direction of agent migration when the agent has the three above mentioned capabilities and when it does not is 26.3KB. This is approximately the size of the three capabilities (24.2KB).

Table 6-5: Summary of IP data traffic as agent Bob migrates without capabilities

Agent Travel Direction	Total		A to B		B to A	
	Packets	Bytes	Packets	Bytes	Packets	Bytes
Laptop (B) to Server5 (A)	96	60426	37	2447	59	57979
Server5 (A) to Laptop (B)	79	57975	51	56187	28	1788

This step shows that a VERSAG agent is able to migrate from one computing device to another and that it is able to discard its carried capabilities when they are no longer required. The discarding of capabilities results in an agent becoming lighter and is beneficial when it has to migrate over unreliable and slow network links.

Step 2: Upload memo to office virtual noticeboard

We now return to our scenario and proceed to instruct agent Bob 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 Figure 6-5 indicate the steps in carrying out this task. (We note that in the figure, differences in time and time format are due to the messages coming from computers with different system times and default 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 to 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 Figure 6-6. The updated XWiki page with Bob's memo is shown in Figure 6-7.

```
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.
```

Figure 6-5: Complete RMC log messages

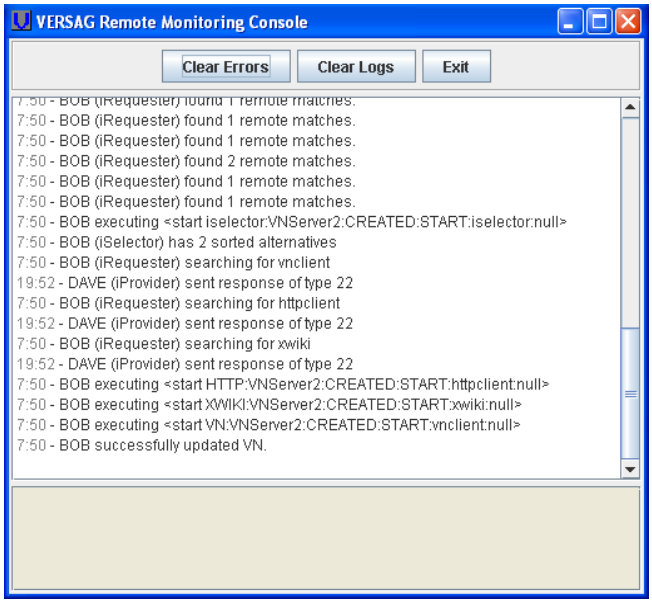


Figure 6-6: Remote Monitoring Console screen capture

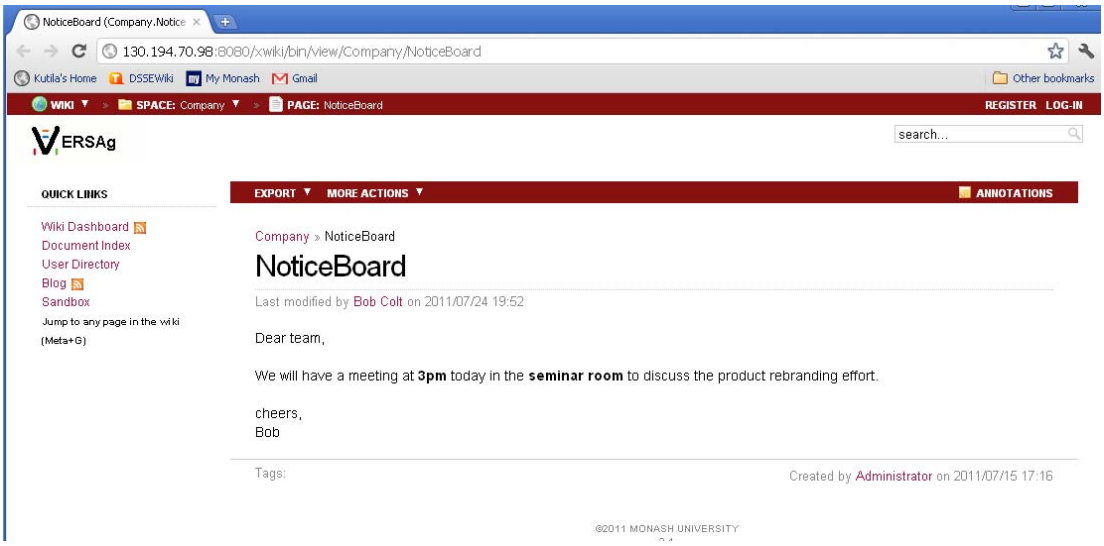


Figure 6-7: Bob's memo on the virtual noticeboard

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 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.

Step 3: 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 6-6. The *get* operations encapsulate VERSAG's cost-efficient capability acquisition process. Bob has earlier identified the applicable cost criteria as network load and time. He has also given a weight of 5 to network load and 4 to time, leading to a relative constraint $con^R = \{5,4\}$ for the cost elements $\{network\ load, time\}$ as in Definition 9 from section 3.3.2.

Table 6-6: 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 (Table 6-2 lists capabilities carried by each agent). This leads to four possible combinations of capabilities (i.e. capability groups). The multi-criteria decision making cost model (proposed in Chapter 4) is used by agent Bob to select the most cost-efficient capability group. The sorted groups and their corresponding utilities as logged by agent Bob are shown in Figure 6-8.

It could be observed that the *2jpodpdfreader,* which is roughly one fourth the size of the *2pdfboxreader* (i.e. 824 and 3176 kilobytes 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. We will further validate this in detail in section 6.4. This therefore illustrates the ability of a VERSAG agent to make a cost-efficient selection based on contextual input and user preferences.

```
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
```

Figure 6-8: Agent Bob's logs showing utilities of sorted capability groups

Agent Bob migrates to *Server6* carrying these acquired capabilities and reuses them there for searching. These capabilities are then discarded prior to migrating to the laptop (*BobLap*) to display results. There again it has to search for a suitable capability to display the results. A screen capture of the result displaying GUI capability presented in Figure 6-9 shows the phrase that was searched for, files containing the search phrase and the word count of each file.

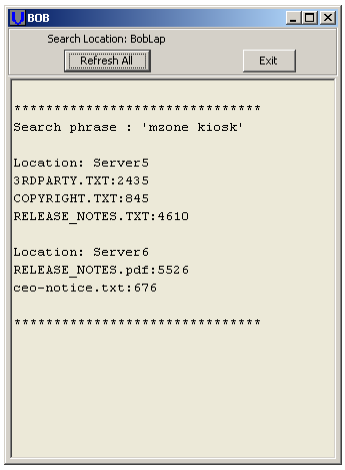


Figure 6-9: Agent displaying search results to Bob on the laptop

The experimental outcomes demonstrate that it is technically feasible to build mobile agents that can perform a given task by dynamically acquiring and discarding functional capabilities. The ability to locate nearby agents and share capabilities with them when required was also demonstrated. Finally, we demonstrated 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.

Having illustrated the benefits of VERSAG in a pervasive computing application scenario, in the following sections we present experimental evaluations validating the individual performance benefits achievable through VERSAG.

6.3 Evaluating Performance Benefits

The main contributions of this thesis include a component-based framework and agent architecture for building compositionally adaptive mobile software agents and the concept of sharing functional components among agents. In this section, we describe two empirical evaluations conducted to establish that our proposed solution can deliver performance benefits, and that VERSAG agents are more suited for dynamic and uncertain environments encountered in pervasive computing. In the two test scenarios we compare the performance of a VERSAG mobile agent with that of a conventional mobile agent (i.e. one that is not compositionally adaptive) identifying their benefits and drawbacks.

6.3.1 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 agents. Thus, our main objective of this experiment is as follows:

- To demonstrate that a VERSAG mobile agent performing a task which requires it to carry out distinct sub-tasks at different locations performs better than a conventional mobile agent assigned with the same task.

Here, we measure performance in terms of overall network traffic generated and time taken to complete a given task/itinerary. These experimental results have been previously reported in [GKL09]. The experimental setup is described below.

Experimental Setting

Bob's personal assistant agent from our example 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, thereby making each task distinct. 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. This operation is also identified as a distinct task. 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.

Figure 6-10 illustrates our test setup to simulate the above scenario. The number of locations is n. 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.

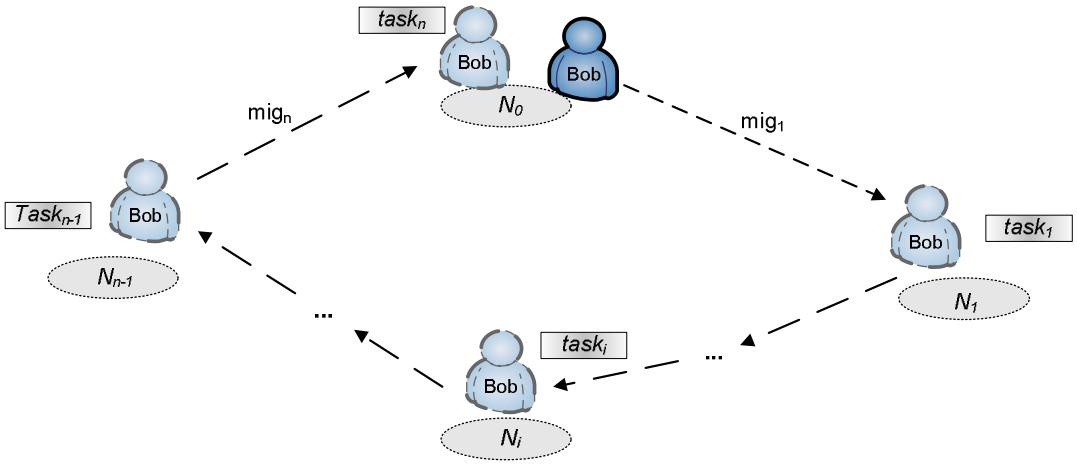


Figure 6-10: Agent migrates to *n* locations, performing a distinct task at each location

We use two types of mobile 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" capability management 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, particular functions required for $task_i$ are implemented as a single capability. In the experimental setup, a separate agent 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 that 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.

As mentioned in chapter 2, the JADE agent platform which we use for testing 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 [BTK05]. 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 [Jad09].

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 normal 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.

Results and Analysis

The mean network loads and itinerary completion times observed during the experimental runs are shown in Table 6-7. A graphical representation of network load variation with number of locations is provided in Figure 6-11.

Table 6-7: Total network load and time for completion (average values)

	Network Load (MB)		**Time (sec)**	
n	**Case I (Conventional)**	**Case II (VERSAG)**	**Case I (Conventional)**	**Case II (VERSAG)**
2	3.53	4.16	1.21	2.14
3	13.87	8.23	4.33	5.83
4	31.20	12.34	7.05	7.97
5	55.23	16.41	12.88	11.02
6	86.35	20.52	17.14	12.23
7	124.10	24.60	24.12	16.96
8	169.03	28.71	33.37	17.24
9	219.83	32.77	42.36	21.50
10	279.18	36.90	50.50	23.49

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 cases.

We assume that overheads for migration and capability exchange (B_{mig}^{ovrhd} and B_{c-ex}^{ovrhd}) 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.

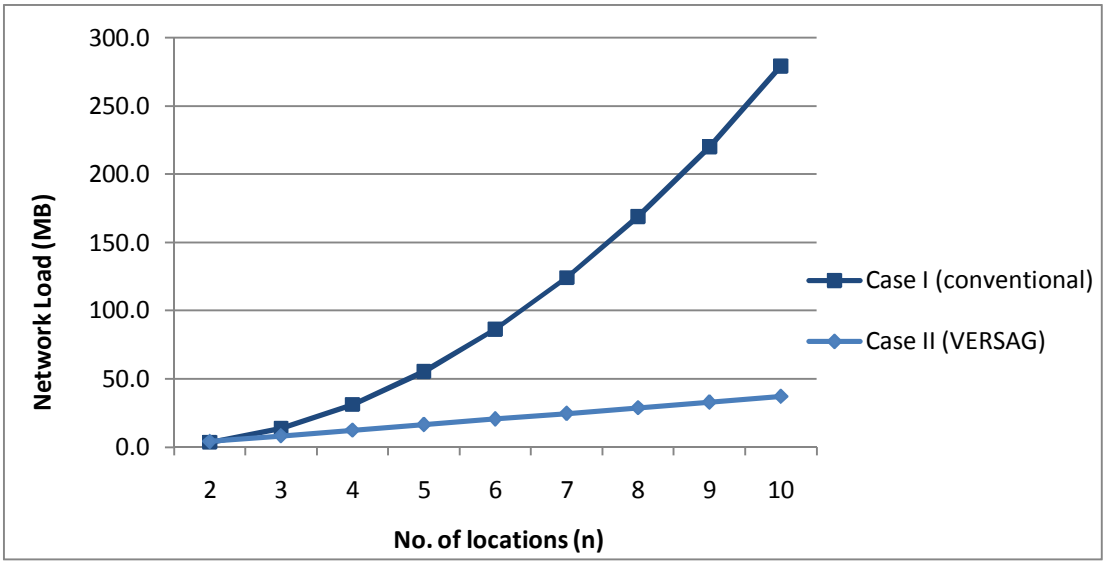


Figure 6-11. Network load Vs No. of locations

Case I

Network traffic generated due to agent migration mig_i can be represented as the sum of agent's class size (B_{MA}^{C}), size of serialized agent (B_{MA}^{Obj}) and migration overheads (B_{mig}^{ovrhd}). 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.

$$B_{mig_i} = \begin{cases} B_{MA}^{C} + B_{MA}^{Obj} + B_{mig}^{ovrhd} & \text{if } i < n \\ B_{MA}^{Obj} + B_{mig}^{ovrhd} & \text{if } i = n \end{cases} \quad (6.1)$$

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

$$B_{MA}^{C} = B_{ma+\sum cap_i}^{C} \quad (6.2)$$

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

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

Expanding this with Equations (6.1) and (6.2),

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

$$B_I = (n-1)B^C_{ma+\sum cap_i} + n\left(B^{obj}_{ma+\sum cap_i} + B^{ovrhd}_{mig}\right) \quad (6.3)$$

Case II

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

$$B_{II} = \left(B_{mig_1} + B_{mig_2} + \cdots + B_{mig_n}\right) + \left(B_{c-ex_1} + B_{c-ex_2} + \cdots + B_{c-ex_n}\right) \quad (6.4)$$

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.

$$B_{c-ex_i} = \begin{cases} B^C_{cap_i} + B^{ovrhd}_{c-ex} & \text{where } i = 1..(n-1) \\ 0 & \text{where } i = n \end{cases} \quad (6.5)$$

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

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

$$B_{II} = (n-1)B^C_v + n\left(B^{obj}_v + B^{ovrhd}_{mig}\right) + B^C_{\sum cap_i} + (n-1)B^{ovrhd}_{c-ex} \quad (6.6)$$

Equations (6.3) and (6.6) enable us to understand the trends displayed by the two plots in Figure 6-11. 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 (6.3) for a conventional agent contains products of the variable terms leading to a quadratic plot while Equation (6.6) 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 the time consumed for itinerary completion, our experimental data presented in Table 6-7 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 mobile 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. This experiment illustrates how VERSAG agents achieve efficient agent migration through peer capability sharing as described in chapter 3.

In our experimental setup, a distinct capability is required at each location. If we relax this condition 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 continues to discard each capability after use, it ends up requesting the same capability multiple times. In such situations, 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. For example, in the next experiment we show that it is beneficial to discard capabilities when travelling over expensive links if they can be acquired at the destination at a lower cost.

6.3.2 Benefits of Localised Capability Sharing

One of the benefits of VERSAG is the ability of an agent to acquire capabilities from nearby peers as opposed to distant suppliers. This brings about both reductions in network load and savings in time. It can also be beneficial in small pockets of agent nodes that have intermittent connectivity with the outside network. Without the ability to share capabilities, agents in such an isolated network pocket may not be able to fulfil certain tasks for which they do not have the necessary capabilities even if other agents in the network already have them. Thus, the main objective of this experiment is as follows:

- To demonstrate that a VERSAG mobile agent can reduce network traffic generated over expensive network links when compared to a conventional mobile agent performing a similar task.

An example of an expensive link is a 3G wireless broadband link which is slower and unreliable compared to a LAN or WLAN. Such networks also incur higher monetary costs as the user is generally charged based on the amount of data transferred. These experimental results have been previously reported in [GZK08a]. We describe the experimental setup next.

Experimental Setting

In this experiment, an agent is assigned a sequential itinerary, which requires it to migrate to 4 locations, performing a different task at each location. The agent originates on a mobile device (N_0) which is connected to the other locations via a poor (i.e. low bandwidth and high latency) wireless network link. The agent path is illustrated in Figure 6-12.

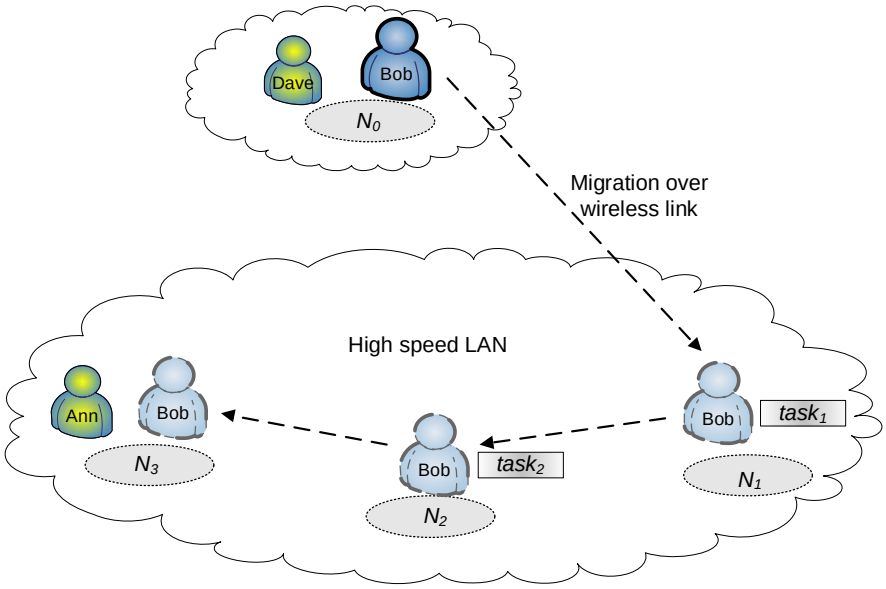


Figure 6-12: Agent migrates from N_0 to N_3 via N_1 and N_2 performing distinct tasks at N_1 and N_2

The tasks used in this experiment consist of sorting a set of text files, at location N_1 using the "insertion sort" algorithm and at location N_2 using the "shell sort" algorithm. We apply the following four solutions to this scenario, using VERSAG agents in three cases and a conventional mobile agent in the fourth case.

Case I: The itinerary is assigned to a VERSAG agent with a "carry-all-with-discard" capability management policy. That is, the agent starts off from location N_0 with all required capabilities. At each location, used capabilities are discarded before migrating to the next location.

Case II: The itinerary is assigned to a VERSAG agent with a "light-travel" capability management policy. That is, it migrates without any capabilities. At N_1 and N_2 it acquires the required capabilities from the peer agent "Dave" which resides at N_0. Thus, capabilities are sent across the wireless link.

Case III: This case is similar to Case II except that the agent acquires capabilities from peer agent "Ann" which resides at location N_3.

Thus, capabilities are acquired from within the local network.

Case IV: A conventional mobile agent with the functions required to execute the two tasks coded into it is assigned the itinerary.

The communication cost (in terms of network traffic) between locations within the LAN are assumed to be negligible while there is a cost involved in data transfer over the wireless network link. We are not concerned with the specifications of the machines or network parameters as our target is to measure the volume of data transferred over the network. The agent platform is restarted between test runs to ensure that any code caching mechanisms do not affect the measurements. Total network traffic generated over the wireless link and time taken to load and execute a capability in each case is measured.

Results and Analysis

The mean observed data volumes and capability execution times for the four cases are shown in Table 6-8.

Table 6-8: Network traffic over wireless link and itinerary completion times

Test Case	**Data transferred (KB)**			**Execution Time (ms)**	
	Total	**N_0 to N_1**	**N_1 to N_0**	**Capability 1**	**Capability 2**
I	360.5	277.3	83.2	309	79
II	389.8	294.9	95.0	312	75
III	134.8	94.6	40.2	316	78
IV	319.0	242.5	76.5	318	78

The execution times for tasks are similar when loaded as capabilities (Cases I, II and III) and when coded into the agents (Case IV), indicating negligible overhead due to the added application level class loading and execution. Figure 6-13 provides a graphical representation of the amount of data transferred over the wireless link for the four test cases.

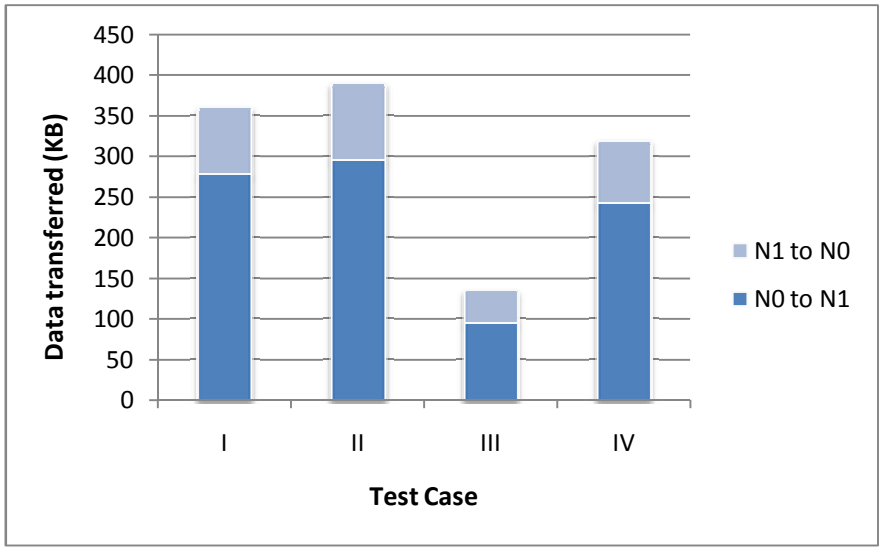


Figure 6-13: Traffic generated over wireless network link in both directions.

It can be observed that more data is transferred from N_0 to N_1 which is the direction in which the agent migrates and code/capability transfers occur. Traffic in the reverse direction is generated due to messages passed during migration/capability exchange and for JADE administration. Cases I and II generate similar amounts of traffic with Case II being slightly higher due to the overheads of searching for and requesting capabilities. In Case IV, the code for capabilities are included in the agent as in Case I; however the size of the agent is smaller than the VERSAG agent. Therefore we see a slight reduction in the total data transferred. Case III shows a significant reduction in the number of bytes transferred since the capabilities are obtained from within the local network (i.e. agent Ann on location N_3).

We model network traffic over the wireless link for the four cases in Equations 6.7 to 6.10. In Case I, it consists of the agent classes (B_v^C) and objects (B_v^{obj}), the two capabilities and migration overheads. For Case II, it is made up of the agent migration, and two capability exchanges which result in additional overheads (B_{c-ex}^{ovrhd}). For Case III, since capability exchange occurs with agent Ann within the LAN, traffic over the wireless link is only due to migration of the agent. Finally, in Case IV, traffic consists of the agent's classes (B_{ma}^C) and objects (B_{ma}^{obj}), classes for functions implemented by the two capabilities and migration overheads. We assume that the classes for the two functions are of the same size as the capabilities.

$$B_I = \left(B_v^C + B_v^{obj} + B_{cap_1}^C + B_{cap_2}^C + B_{mig}^{ovrhd}\right) \quad (6.7)$$

$$B_{II} = \left(B_v^C + B_v^{obj} + B_{mig}^{ovrhd}\right) + \left(B_{cap_1}^C + B_{c-ex}^{ovrhd}\right) + \left(B_{cap_2}^C + B_{c-ex}^{ovrhd}\right) \quad (6.8)$$

$$B_{III} = \left(B_v^C + B_v^{obj} + B_{mig}^{ovrhd}\right) \quad (6.9)$$

$$B_{IV} = \left(B_{ma}^C + B_{ma}^{obj} + B_{cap_1}^C + B_{cap_2}^C + B_{mig}^{ovrhd}\right) \quad (6.10)$$

The above experimental results and network traffic models demonstrate that a mobile agent enhanced with peer capability sharing can reduce network traffic generated over a specific (expensive) network link when compared to a conventional mobile agent performing a similar task. The agent achieves this by reusing code available at nearby locations.

Since capability sharing incurs an additional overhead as seen above, VERSAG agents need to make use of this feature efficiently and intelligently. We proposed the capability acquisition decision making cost model as one mechanism to improve an agent's cost-efficiency. In the next section, we present experiments to evaluate the performance of this cost model.

6.4 Evaluating Cost Model Performance

The third evaluation criterion we investigate is the performance of the decision making cost model proposed in chapter 4. This section describes two sets of experiments carried out specifically for this purpose. The first set of experiments is targeted towards validating the accuracy of the cost model while the second set of experiments illustrates how the cost model can be used to increase the efficiency of the agents.

6.4.1 Accuracy of the Decision Making Cost Model

The objectives of our first set of experiments are stated as follows.

1. To demonstrate that when the agent has multiple alternative capabilities which it can acquire in order to fulfil a given activity, it selects the least cost alternative.

2. To demonstrate that the internally computed cost estimates used in the cost model are representative of the actual costs.
3. To demonstrate that the cost savings gained by using the cost model override overheads of cost estimation.

We describe two sets of experiments: one using *time* consumed as the cost criterion and the other using *network load*. No absolute constraints are specified in both cases. These experimental results have been previously reported in [GKL10].

Experimental Setting

A VERSAG agent is assigned an itinerary which requires it to carry out a workload for which the needed capabilities have to be searched and acquired from peer agents. The experiment does not require agent migration. We have selected three separate workloads representing:

- A sample agent GUI,
- An information retrieval task, and
- A file sorting task.

For each workload, two possible alternatives could be formed from capabilities available with peer agents. Thus, for each test, the agent has to decide on the least cost alternative.

Tests are conducted over a high-speed LAN and an IEEE 802.11g wireless network while a 3G wireless broadband link is also used for load based tests. We use three test computers running Windows XP, Windows Vista and Ubuntu Linux operating systems. All have Java SDK 1.6.0 and VERSAG is implemented on the JADE agent platform version 3.7 with the LEAP add-on. The network protocol analyser Wireshark [LSW11] is used to measure the actual network load generated and time consumption is recorded through code instrumentation.

For each workload, the agent selects a "least cost"alternative. For this particular alternative, we then repeat the tests with artificially high execution times/loads manually set (in capability meta-data). This results in the agent selecting the

previously ignored alternative and allows us to measure the actual performance of both alternatives for comparison purposes.

Equations (4.3) to (4.8) in chapter 4 presented the formulae used by the agent for estimating time and network load. The time taken when searching for capabilities and applying the cost model (T^{search}) and load generated during searching for capabilities (B^{search}) were not included in the agent's estimation since they are common for all available alternatives. However, they are useful in understanding overheads of the selection process. Therefore, they are estimated separately and included in the results reported in this sub-section.

Every prototype agent is equipped with a context-sensing capability which enables it to measure network bandwidth and latency. Other network related parameters, namely capability request size (B^{creq}), request/response overhead (B^{ovrhd}), decision making load (B^{search}) and decision making time (T^{search}) are estimated manually using trial runs and supplied to the context-sensing capability.

For the IEEE 802.11g wireless network, the context-sensing capability measured average latency (λ) as 1ms and average bandwidth (bw) available to agents as 13.5Mb/sec. For the LAN, latency was negligible and average available bandwidth was 81.3Mb/sec. Based on our trial runs it was seen that B^{search} is proportional to the number of peer agents to be searched in, and it was estimated accordingly. Similarly, request size and message overhead were estimated to be 224 and 2000 bytes respectively while decision making time (T^{search}) was estimated to be 300 milliseconds. It is expected that in real-life situations the context-sensing capability would make these estimations. Of the needed capability meta-data, capability sizes (B_i^{cap}) were measured by the peer agent. Time to start a capability ($T_i^{start\_p}$) and execution time (Tp_i^{loc}) were also estimated in trial runs and added to capability meta-data. For the experimental workloads, network traffic generated during execution (B_i^{exec}) was nil.

Results and Analysis

With the above setup, the tests were run multiple times and observations recorded. In Table 6-9, we show estimated utility values for the three workloads with actual measured costs. We note that the higher utility is indicative of lower cost and is the preferred alternative. The two available alternatives for each workload are identified as *a* and *b*. Columns 3 and 4 are for tests with *time* as the constraint and columns 5 and 6 are for tests with *network load* as the constraint.

Table 6-9: Predicted utility values and actual costs for the three test workloads

Workload	Alternative	Time		Network Load	
		Utility	Actual (ms)	Utility	Actual (KB)
1	a	**0.78**	709	0.02	731.0
	b	0.22	1089	**0.97**	17.8
2	a	**0.75**	1306	0.21	3495.9
	b	0.25	2919	**0.78**	913.5
3	a	**0.90**	638	0.4995	11.2
	b	0.10	1378	0.5004	11.4

We see that when time is the cost criterion all predictions are accurate with the higher utility alternative consuming less time than the other. When network load is the cost criterion, for the first two workloads the agent correctly estimates higher utility values while in workload 3, the higher cost alternative is selected. However, in this case the utilities are indistinguishable up to three decimal places and the difference between observed costs of the two alternatives is less than 200 bytes. Thus, it can be safely considered that the difference between the two alternatives is sufficiently small and has no significant impact on performance. These results clearly indicate that the cost model is able to correctly select the least cost alternative for the agent to fulfil its goals.

The second objective of these experiments is to verify whether the cost estimates are representative of the actual costs. For this purpose, we plot the cost estimates together with the actual measured values from experimental outcomes. Figure 6-14 shows network load estimates for 18 experiments (split into two graphs due to differences in estimated value ranges) and Figure 6-15 shows time consumption.

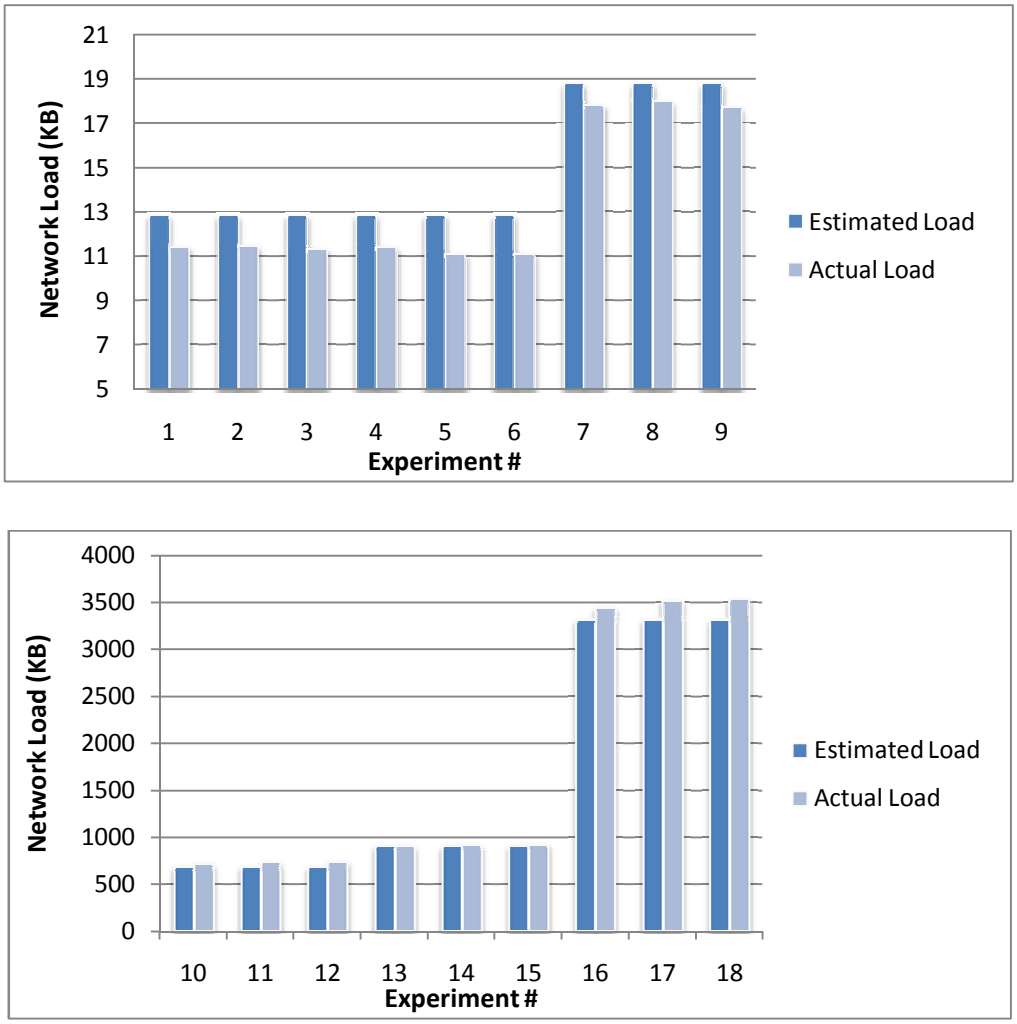


Figure 6-14: Comparison of Actual and Estimated network loads sorted in ascending order of estimate

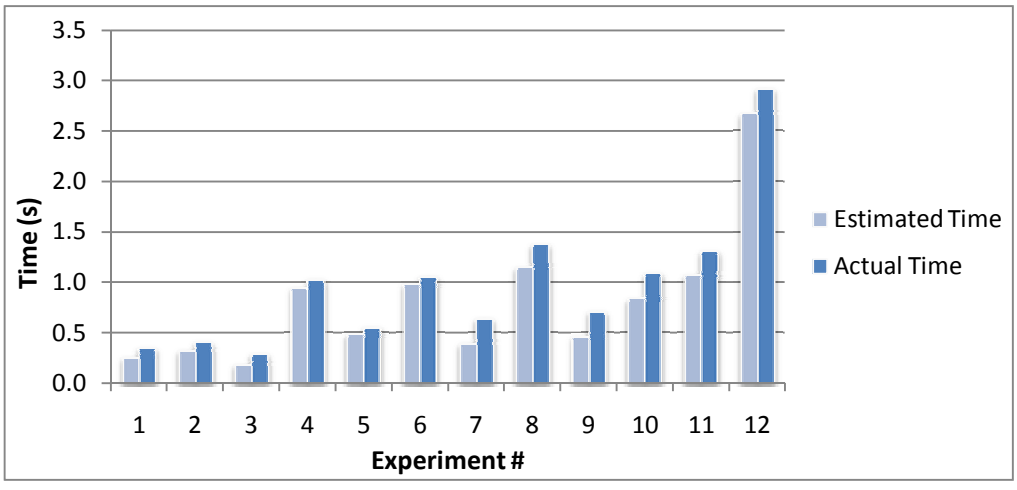


Figure 6-15: Comparison of average Actual and Estimated time consumption

The graphs clearly illustrate that our estimates are representative of actual costs, an indication that the strategies used to estimate time and network load are accurate. Since they are used by the cost model to make adaptation decisions, it is essential that the estimates are accurate. The percentage error between estimated and actual values as a percentage of the estimated value was calculated as follows:

$$percentage\ error = \frac{actual\ cost - estimated\ cost}{estimated\ cost} \times 100$$

For the network load based experiments, mean percentage error was 7.17% with a maximum of 13.5%. For time based experiments, error percentages were less than 10% when the estimate was larger than 200 milliseconds while higher error percentages were observed for smaller estimates.

The third objective of the experiments was to demonstrate that the cost savings gained by using the cost model override overheads of cost estimation. The measured overhead due to the adaptation cost model and decision process was less than 4KB in terms of network load and 300 milliseconds in terms of time consumption. Thus, the cost model improves agent efficiency when savings achievable through the use of the cost model are higher than these values, which are relatively small overheads.

While our experimental results validate the accuracy of the cost model, prediction accuracy is dependent on multiple factors, the principal one being the meta-data provided by capability developers. The time to start and execute a capability and network traffic generated by the executing capability are two such examples of meta-data. These in turn depend on other factors such as the runtime environment and data being processed. As outlined in chapter 4, we envision that a capability's temporal and network load generation behaviours will be tested on a standard environment and used to obtain the meta-data. These can then be adapted by the agent to suit its current environment. Another factor which can affect the prediction accuracy is the non-deterministic nature of the environment, such as variations in network parameters and processing load. Thus, while our cost model itself is accurate for adaptation decisions, it is dependent on the quality of the estimates for the individual cost components.

6.4.2 Efficient Service Oriented Agents with VERSAG

We now describe an experiment where the cost model is used to increase the time efficiency of service-oriented software agents. While thus far we have focused on peer agents providing capabilities, it is also possible to envision the acquisition of capabilities through remote web services or to invoke capabilities on remote web services. We see this as an illustration of the flexibility and extensibility of VERSAG.

The objective of this section is:

- To demonstrate how the adaptation decision making cost model of VERSAG can be used to increase the efficiency of service-oriented software agents through dynamic selection of processing schemes.

The experimental evaluations and outcomes described in this sub-section have been previously reported in [GLZ11].

The Scenario

We consider a scenario of an information gathering mobile agent in a pervasive computing environment. The agent is required to migrate to a number of locations (i.e. computing devices) and process various types of data found on them. Two examples of processing are: to extract information from a set of data files where the processing required depends on the file type (e.g. TXT, PDF, DOC); and to read data generated by sensors on the device and extract information. Thus, the processing depends on the file types or on the sensor interface and nature of the information extraction task. Since processing requirements are not known in advance, the agent does not contain the necessary processing logic. Once the required processing is identified, the typical approach is for the agent to select and invoke an appropriate remote web service or service agent to process the data.

We hypothesize that in mobile computing environments it can be beneficial if the software agent, in addition to invoking remote services, can acquire the capacity to execute functionality provided by the service and run it locally. Such an alternative is beneficial when the cost of service invocation is significantly higher compared to

that of running the service locally. For example, in the context of exploratory mobile agents for information gathering in a disaster zone as described in chapter 1, invocation of remote web services or service agents to process the different types of data they encounter is prohibitive due to unreliable connectivity and low resource availability. Therefore we assume that the agent should have the choice of downloading a software component which implements the required processing logic and executing it locally. VERSAG is ideally suited to implement such agents. Next, we describe our experimental setup to illustrate and evaluate this scenario.

Experimental Setting

For the experiments, we consider an agent that has already migrated on to a portable computer and has to process a large number of text files located on it. As per our scenario, it is assumed that the agent is unaware of the type of files prior to arriving at the device. Therefore, it has to select an appropriate capability with which to fulfil the task after arriving at the device. The agent has two capabilities to select from, leading to the following two cases (The two cases are illustrated in Figure 6-16).

Case I The agent selects a web service client capability which invokes a remote (web) service to process the data. For each datum, a service request encapsulating the datum is sent and a result obtained until all the data are processed.

Case II The agent acquires a capability which has the know-how to process the data from a remote provider. The agent then repeatedly executes this capability locally to process the data.

The processing involved is to convert the files to PDF format. For Case I, the agent already has the capability that allows it to access a SOAP (Simple Object Access Protocol) based *txt2pdf* web service that can do the conversion. The web service accepts a single text file in a SOAP request, converts it to a PDF and sends it back to the client encapsulated in a SOAP response. The agent has to sequentially request the web service to convert files. For Case II, a *txt2pdf* capability is available with a peer agent at the same location as the *txt2pdf* web service.

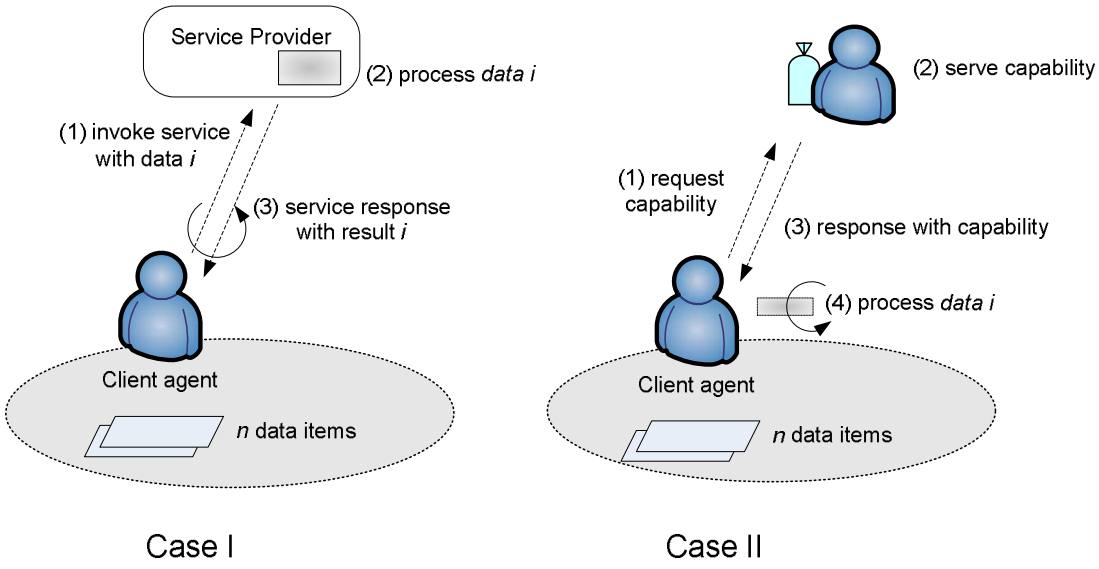


Figure 6-16: Service use Vs Component use to process data

To make the appropriate selection, the agent makes use of a *scheme-selector* capability which is an enhanced version of the decision making cost model described in chapter 4. The measure of efficiency is task completion time (i.e. the *relative constraint* is time).

Inputs needed for scheme selection are network parameters, web service SLAs (Service Level Agreements) and details of data (i.e. type, size, speed of generation). Network information is obtained from a separate context-monitoring capability while the web service client capability provides web service parameters. Based on this information, the *scheme-selector* decides whether it is more efficient to use the web service or process locally.

Our test environment is as follows. The web service is deployed on a desktop PC with Windows XP, and the client agent runs on a slower notebook PC with Windows Vista. Both computers have Java SDK 1.6.0 and the agent toolkit used is JADE v3.7 with the LEAP add-on [BCG07]. The desktop PC is connected to a LAN (Local Area Network) which has high-speed Internet connectivity (1Gbps). The notebook PC is connected first over an IEEE 802.11g wireless local area network (WLAN) and subsequently over an HSDPA (High-Speed Downlink Packet Access) network. The measured network latency in the WLAN environment was less than 0.5ms and the observed bandwidth 15Mbits/sec. Over the HSDPA network, the average latency

was 158ms and observed average bandwidth 300Kbits/sec. The size of the *txt2pdf* capability is 1070KB.

We test time taken to complete the processing as the size of data increases. Total file size is increased from 250KB to 3000KB while keeping the number of files fixed at 10. Time measured for Case I (web service use) is from the start of execution of the client capability to its completion. In Case II (component use), since the agent has to acquire the capability, we also measure time taken to acquire the capability in addition to capability execution (i.e. local data processing) time. Time taken for scheme selection is not included. Network load generated is also recorded and reported.

Results and Analysis

When the experiment was conducted on the WLAN network, the *scheme-selector* consistently selected the web service (Case I) approach while on the HSDPA network it selected the component use (Case II) approach for the majority of the data sizes. Table 6-10 shows utility values computed for 3000KB data on the two networks.

Table 6-10: Scheme-selector computed utility values for 3000KB data files

Network	Utility of web service use (I)	Utility of component use (II)
WLAN	0.58	0.42
HSDPA	0.29	0.71

The experiments were then repeated by altering the agent's itinerary to bypass the *scheme-selector* and pre-select an approach. We plot the observed variation in response time as total data size increases for both approaches in Figure 6-17 and Figure 6-18. For both graphs, number of files is 10 and average values are used to plot graphs.

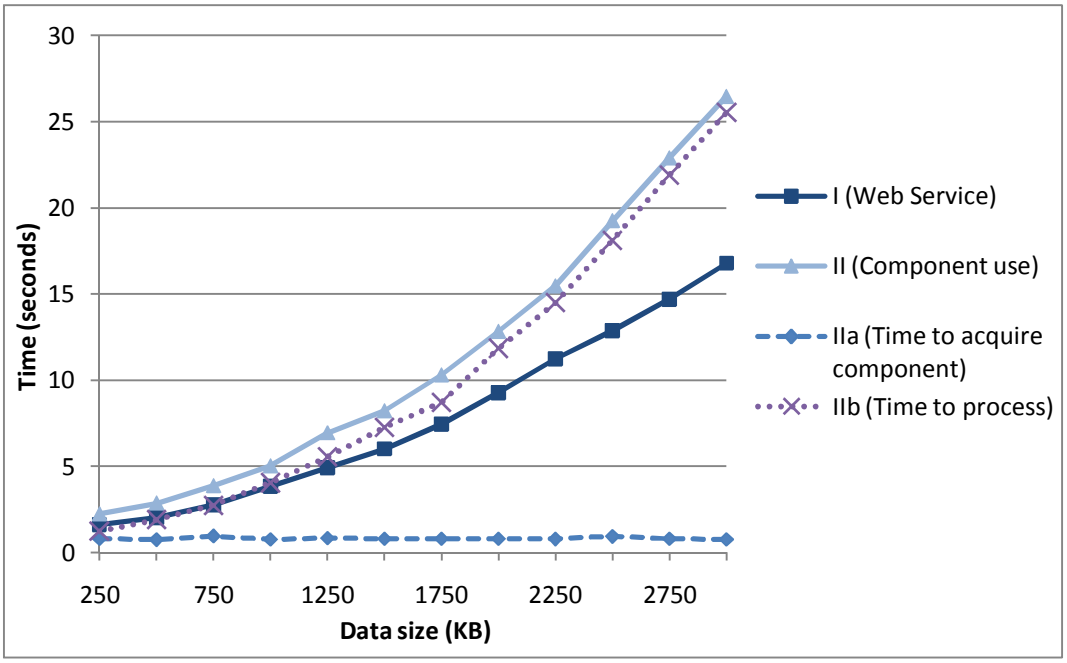


Figure 6-17: Response time variation with data size (WLAN network)

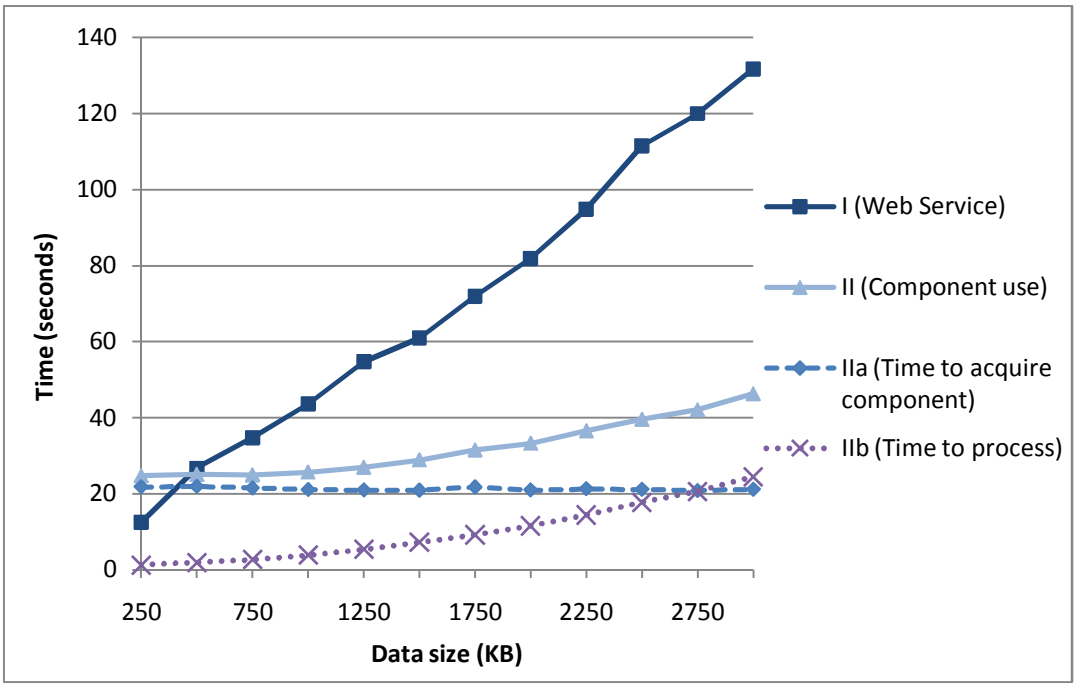


Figure 6-18: Response time variation with data size (HSDPA network)

Figure 6-17 is on the WLAN network, and shows that web service (I) approach consistently consumes less time than the component use (II) approach. Figure 6-18 for the HSDPA network shows that the component use (II) approach consumes less time for all cases except 250KB data size. The behaviour of these plots confirms that the *scheme-selector* correctly predicted and selected the least cost approach.

To further aid analysis of these observations we build an analytical model of time consumption and network load for the two cases. We assume that overhead of each request/response is a constant value B^{ovrhd}. The start up time of the data processing component at the agent's site (T^{start_p}) is fixed but non-negligible and is therefore included in the model.

Case I

The time to process a single datum consists of the time to send the request to the server ($T_i^{serv\_req}$), time taken to process the request at the server (Tp_i^{svr}) and time to send the result back to the agent ($T_i^{serv\_res}$). Time taken to process n data items is the sum of time taken to process each datum.

$$T_I = \sum_{i=1}^{n} \left(T_i^{serv\_req} + T_i^{serv\_res} + Tp_i^{svr} \right) \tag{6.11}$$

From Equation (4.4) we have,

$T_i^{serv\_req} = \lambda + \frac{\left(B_i^d + B^{ovrhd}\right)}{bw}$ and $T_i^{serv\_res} = \lambda + \frac{\left(B_i^{res} + B^{ovrhd}\right)}{bw}$.

Here, λ and bw represent latency and bandwidth of the network. The size of data item *i* is B_i^d and B_i^{res} represents the size of the result from processing data item *i*. Tp_i^l shows the time needed to process data item *i* at location *l*. Substituting these in Equation (6.11) we get,

$$T_I = 2n\left(\lambda + \frac{B^{ovrhd}}{bw}\right) + \frac{1}{bw}\sum_{i=1}^{n}\left(B_i^d + B_i^{res}\right) + \sum_{i=1}^{n} Tp_i^{svr} \tag{6.12}$$

Case II

The total time in Case II is made up of the time to request (T^{cap_req}) and retrieve (T^{cap_res}) the component, time to start the component (T^{start_p}) and the time to process (Tp_i^{loc}) all the data items locally.

$$T_{II} = T^{cap\_req} + T^{cap\_res} + T^{start\_p} + \sum_{i=1}^{n} Tp_i^{loc} \tag{6.13}$$

From Equation (4.4) we have,

$T^{cap\_req} = \lambda + \frac{(B^{creq} + B^{ovrhd})}{bw}$ and $T^{cap\_res} = \lambda + \frac{(B^{cap} + B^{ovrhd})}{bw}$.

Here B^{cap} is the size of a component and B^{creq} is the size of a command requesting a component. Substituting these in equation (6.13) we get,

$$T_{II} = 2\left(\lambda + \frac{B^{ovrhd}}{bw}\right) + \frac{(B^{creq} + B^{cap})}{bw} + T^{start\_p} + \sum_{i=1}^{n} Tp_i^{loc} \tag{6.14}$$

Network traffic generated between the agent and service provider in Case I is made up of requests sent by the agent, and responses received. Each request encapsulates a datum while the response contains the corresponding result. Together with the messaging overheads, this is expressed in Equation (6.15). For Case II, generated network traffic consists of only the request from the client and the corresponding response encapsulating the component, as shown in equation (6.16).

$$B_I = 2nB^{ovrhd} + \sum_{i=1}^{n} (B_i^d + B_i^{res}) \tag{6.15}$$

$$B_{II} = 2B^{ovrhd} + B^{cap} + B^{creq} \tag{6.16}$$

In Figure 6-17 (WLAN), response times for both web service (I) and component use (II) approaches start at similar levels but the component use approach increases more rapidly. Examining the component use (II) time breakdowns in (IIa) and (IIb), we observe that data processing time is the most significant component. This behaviour is consistent with Equations (6.12) and (6.14) where it can be seen that in a network with high bandwidth (*bw*) and low latency (λ) such as the WLAN, data processing time ($\sum_{i=1}^{n} Tp_i^{loc}$ or $\sum_{i=1}^{n} Tp_i^{svr}$) is the significant component. In the graph, it can also be seen that data processing time for component use (IIb) is higher than the total time for web service use (I). This is due to differences in processing speeds of the two computers used. When the web service was deployed on the notebook computer, it was observed that web service use (I) time increased significantly (e.g. from 18.4 seconds on desktop PC to 30.7 seconds on notebook for 3000KB of data) such that its performance was far worse than the component use (II) approach.

In a low bandwidth, high latency network, the components of data size and component size come into consideration as seen in Equations (6.12) and (6.14) respectively. Furthermore, for component use (Case II) there is no direct influence from increasing data size, except that it leads to increased processing times. Figure 6-18 shows response time variation when the tests were run on the HSDPA network, which is a low bandwidth high latency network. We observe that the web service (I) approach consumes increasingly more time with increasing data size compared to the component use (II) approach. Component acquisition (IIa) consumes more time than processing (IIb) initially, but remains constant and is overtaken by the processing time as data size increases. In the web service (I) approach where each data file has to be transferred, time consumption increases more rapidly with total data size while increases in the component use approach occur only as a result of increases in processing time.

We further used the Wilcoxon signed-rank test to test the statistical significance of time consumption in the two approaches. Our null hypothesis (H_0) is that there is no significant difference between time consumption in the two approaches. The alternate hypothesis (H_1) for the WLAN network is that the component use approach consumes more time than the web service use approach. The null hypothesis was rejected at a significance level of 0.005. Therefore, it can be stated with a confidence value of 99.5% that the component use approach consumes more time than the web service approach on the WLAN network. For the HSDPA network our alternate hypothesis (H_1) states that the component based approach improves efficiency by reducing time consumed. Here too the null hypothesis was rejected at a significance level of 0.005. Therefore, it can be stated with a confidence value of 99.5% that the component use approach consumes less time than the web service approach when tested on the HSDPA network.

Further examining the two experiments, in Figure 6-19 and Figure 6-20 we show network load variation corresponding to the two test cases. It should be noted here that network load was not used as a criterion for scheme selection. It is however evident that traffic generated in the component use (II) approach is independent of data size as data items are not moved. In contrast, for the web service (I) approach

data has to be transferred to be processed and results retrieved. Therefore generated load increases with data size. This behaviour is common across both networks (WLAN and HSDPA) and is consistent with observations made based on Equations (6.15) and (6.16).

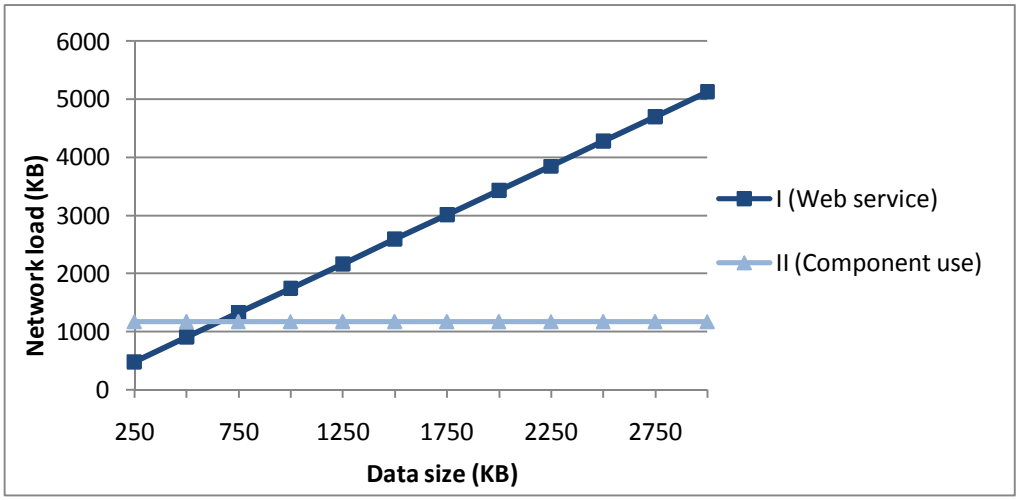


Figure 6-19: Network traffic in WLAN network with increasing data size

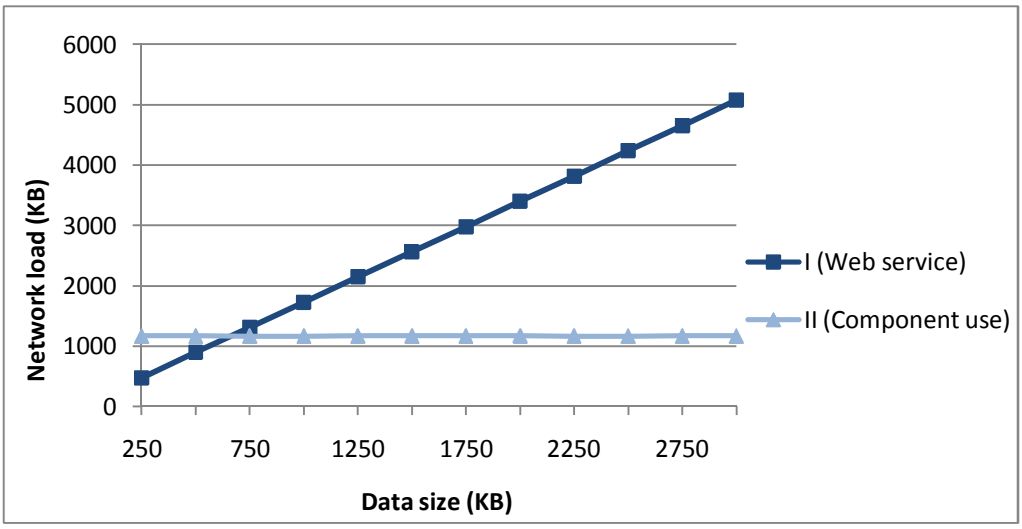


Figure 6-20: Network traffic in HSDPA network with increasing data size

Based on these experimental results it can be concluded that on a low bandwidth network, as the size of data to be processed increases, the component use approach becomes more attractive than the web service use approach since there is no need to transfer the data. With regards to network traffic generation, the component use approach generates a constant load as it moves the code to where the data is, while in the web service case, data has to be transferred to where the code resides. Thus, once

the data size increases beyond the size of the code, the component use approach becomes preferable irrespective of the network type. Another factor that contributes to such decision making, which was not considered in our experiment, is the computational capacity of the computing devices.

Thus, given information about its environment, including bandwidth, computational capacity of each location, size of data to process as well as size of the capability, the agent can dynamically choose a better processing scheme to improve its efficiency in terms of relevant criteria. Furthermore, if the agent had chosen the web service scheme to start with, but during processing the network performance degrades (e.g., the device moves from a WLAN to a wide-area lower bandwidth connection), the agent could switch schemes in order to maintain performance levels. Thus, the above experimental results and analysis demonstrate and validate how the adaptation decision making cost model of VERSAG can be used to increase the efficiency of service-oriented software agents.

Having validated the functional and performance benefits of VERSAG and its cost model, in the next section we investigate how VERSAG scales as the number of agents and locations increases.

6.5 Evaluating VERSAG Scalability

As identified at the beginning of this chapter, the usefulness of VERSAG is maximized when there is a large distributed ecosystem consisting of agents engaged in diverse activities and interacting with each other. Therefore, it is necessary for our agents to be able to execute in environments with large numbers of agents. The purpose of this section is to evaluate the performance of VERSAG in situations that can occur as the number of agents in the environment increases.

Since our prototype is implemented on top of the JADE agent platform, many of its performance and scalability characteristics are inherited by VERSAG. For example, we use communication, migration and directory services provided by JADE. The performance of JADE, covering many of these features, has been evaluated and reported in previous research (e.g. [ASF10, CGK05, MSA06, SAM07]). Therefore

we focus our evaluations in this section on factors that are specific to the VERSAG framework. Furthermore, in all the experiments we use time as the performance measuring criterion.

6.5.1 Multiple Co-located VERSAG Agents

Our first experiment investigates the ability of multiple VERSAG agents to co-exist at a single location (i.e. JADE container). A similar experiment, simulating an e-commerce application scenario was used to evaluate the JADE platform in [CGK05]. The objective of this experiment is as follows:

- To investigate the ability of a large number of VERSAG agents to co-exist at a single location (i.e. JADE container) without causing any errors or significant performance degradation.

Experimental Setting

In this experiment, as illustrated in Figure 6-21, we simultaneously instruct a number of agents to migrate to a particular location, do some processing and migrate out of the location. The workload assigned is to sort the content of a text file (412KB) using a "shell sort" capability. The agent already possesses the required capability and there is no capability sharing involved in this experiment.

The location for deployment is a personal computer with an Intel Pentium IV 3.0GHz processor, 1GB memory and running version 9.10 of Ubuntu Linux Operating System. The software environment consists of Java SDK 1.6.0 and JADE version 3.5 with the LEAP add-on. The computer is connected to the other nodes of the agent platform over a high-speed local area network (1.0 Gbps).
The experiment is conducted by increasing the number of agents from 1 to 500. The time taken by each agent to start the "shell sort" capability (i.e. install and start the corresponding OSGi bundle) and to complete the sorting task is measured through code instrumentation.

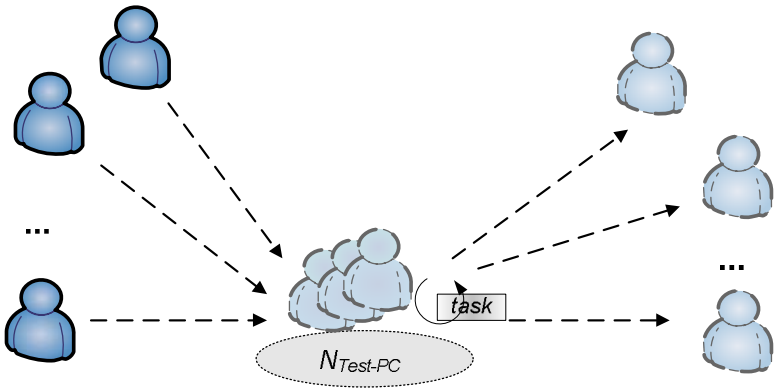


Figure 6-21: Multiple agents on a single location

Results and Analysis

For our experiment, the mean observed times are shown in Table 6-11 and in Figure 6-22. It can be seen that as the number of co-located agents increase, the per-agent start and process times increase. We see a linear increase of time with the number of agents until 300 agents. With 400 agents and above, the test computer reported near 100% CPU and memory usage. This corresponds to the rapid increase in time that can be observed for 400 and 500 agents. The test runs completed in under 1 minute for up to 100 agents while with 500 agents it took over 4 minutes. Further, it was observed that as the agent count increased, agents sometimes failed to migrate out of the location after processing indicating errors in execution (The agents are programmed to discontinue itinerary execution if an error is encountered).

Table 6-11: Time consumption of a single agent as number of agents increases

Number of agents	Time for an agent to start capability (ms)	Time for an agent to work (ms)
1	43	166
10	377	813
50	2,520	3,001
100	5,111	6,105
200	9,641	12,366
300	16,643	18,162
400	32,809	49,686
500	73,469	58,186

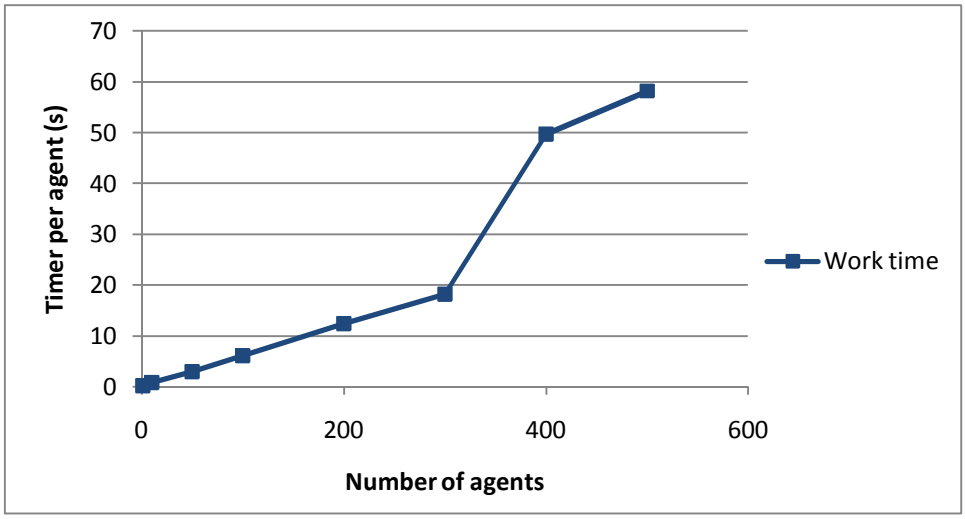


Figure 6-22: Time consumption of a single agent as number of agents increases

Thus, on our test hardware, it is possible to flood an agent location with up to 300 VERSAG agents without causing any errors or significant performance degradation. In comparison, we note that more than 1000 simple JADE agents could successfully flood an agent location. These agents however lack the enhanced features and functionality of VERSAG.

This observation is acceptable because, unlike a standard JADE agent, a VERSAG agent is multi-threaded and also contains an OSGi container within it. The linear increase of time suggests that it is possible to support higher numbers of agents by increasing resources. The caveat to note is that performance time and CPU consumption will certainly depend on the complexity of the processing task in addition to the number of agents.

6.5.2 Multiple Alternatives for Decision Making

Our second scalability experiment is focused on the decision making cost model of VERSAG and is similar to the experiment presented in section 6.4. The specific objective is:

- To investigate the performance of the cost model when an agent has to consider a large number of alternatives for decision making.

Experimental Setting

In this experiment, as illustrated in Figure 6-23, a VERSAG agent ("Bob") is assigned an itinerary which requires it to search and acquire a capability from its peer agents. The experiment does not require agent migration. We increase the number of peer agents in the agent platform that contain the required capability, thereby increasing the number of alternatives that the agent has to consider in its decision making process. The capability seeking agent is deployed on the same test computer as used in the previous experiment. Potential capability supplying agents are deployed on separate computers.

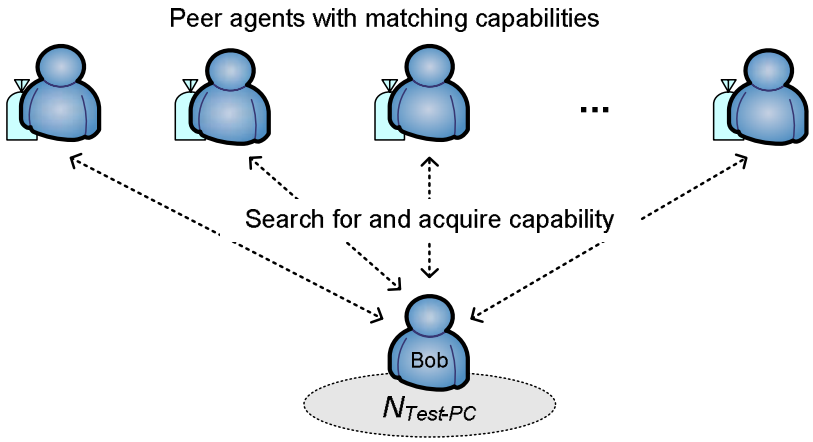


Figure 6-23: Multiple alternatives for decision making

The experiment is conducted by increasing the number of peer agents from 1 to 100, each agent containing 5 matching capability instances. Time taken by the capability seeking agent to make a capability selection decision after initial search responses were received from all agents is measured. Decision making time therefore consists of time taken to estimate the relevant cost components, aggregate and sort them using the cost model presented in chapter 3. Time taken to subsequently acquire the selected capability is also measured.

Results and Analysis

The observed decision making times and capability acquisition times are shown in Table 6-12 while Figure 6-24 plots the behaviour of decision making time as the number of alternatives increases.

Table 6-12: Decision making and acquisition times with increasing number of alternatives

No. of peer agents	No. of alternatives	Decision making time (ms)	Acquisition time (ms)
1	5	333	206
2	10	557	164
10	50	2393	232
20	100	4624	251
40	200	9143	403
60	300	13784	416
80	400	19067	676
100	500	29437	727

We observe a rapid and linear increase in the decision making time as the number of available alternatives increases. While the exact impact of this overhead depends on the nature of the task to be executed (i.e. task execution time), it is indicative of the need to limit the number of alternatives that an agent considers in its decision making steps. Two other approaches with which the impact of the decision making process on the agent can be reduced are: offloading the decision process to an idle agent or controlling application; and ignoring low impact cost elements (e.g. if relative importance is less than 1%).

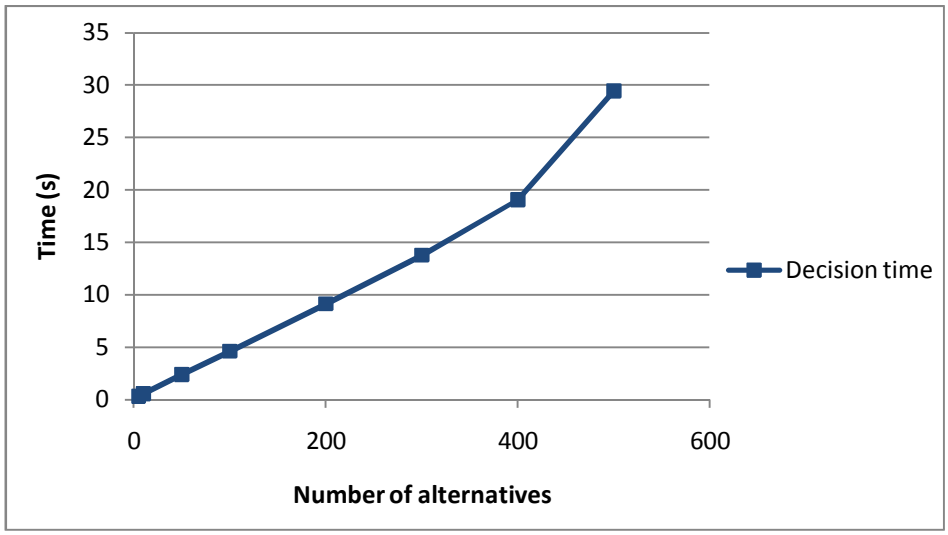


Figure 6-24: Decision making time with increasing number of alternatives

6.5.3 Multiple Capability Requests to a Single Agent

In our third experiment on scalability, we investigate how well a single VERSAG agent can serve multiple simultaneous capability requests from peer agents. The objective of this experiment is as follows:

- To investigate the performance of peer capability sharing when a large number of agents are requesting capabilities from a single agent.

Experimental Setting

As illustrated in Figure 6-25, in this experiment a single agent is flooded with capability requests from a large number of agents.

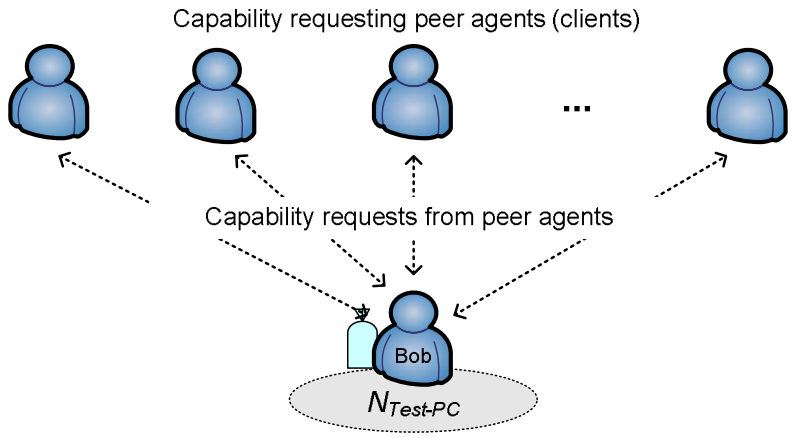


Figure 6-25: Single agent flooded with multiple capability requests

The client agents first search for matching capabilities, and then send a request for a capability. Therefore, our capability supplier agent ("Bob") receives two requests from each client agent. The capability supplier agent is deployed on a personal computer with an Intel Core 2 Duo E6300 1.86GHz processor, 2GB memory and running Windows XP Operating System. Client agents are deployed on the test computer running Ubuntu Linux. Agent "Bob" is not assigned any other itinerary and is able to use all its resources for serving peer capability requests. The experiment is conducted by increasing the number of client agents from 1 to 50. Time taken by each agent to send a capability request and receive a capability in response is measured. That is, we do not measure time taken for the initial search requests sent.

Results and Analysis

The observed average, minimum and maximum capability acquisition times in client agents is shown in Table 6-13 and the average times plotted in Figure 6-26.

Table 6-13: Capability acquisition time with increasing number of client agents

Number of client agents	Time for a client agent to acquire capability (ms)		
	Average	Minimum	Maximum
1	158	133	222
5	248	163	313
10	279	164	426
15	457	168	1668
20	473	187	2659
25	628	223	1894
30	843	256	2757
35	1017	259	3083
40	1287	251	3639
45	1546	329	4445
50	1654	272	5,180

Within the tested range, the capability supplier agent could successfully respond to all capability requests. However, once again we observe a nearly linear increase in acquisition time as the number of requesting agents increase. Therefore, a heavily loaded capability supplier agent can result in poor capability sharing performance of its client agents. Furthermore, as mentioned in chapter 3, the capability sharing module of an agent is expected to have low priority as it is a secondary task of an agent. Thus, from the point of view of the capability supplier agent, it is undesirable to have to serve too many capability requests. Therefore, VERSAG agents need to restrict the amount of peer capability requests they handle.

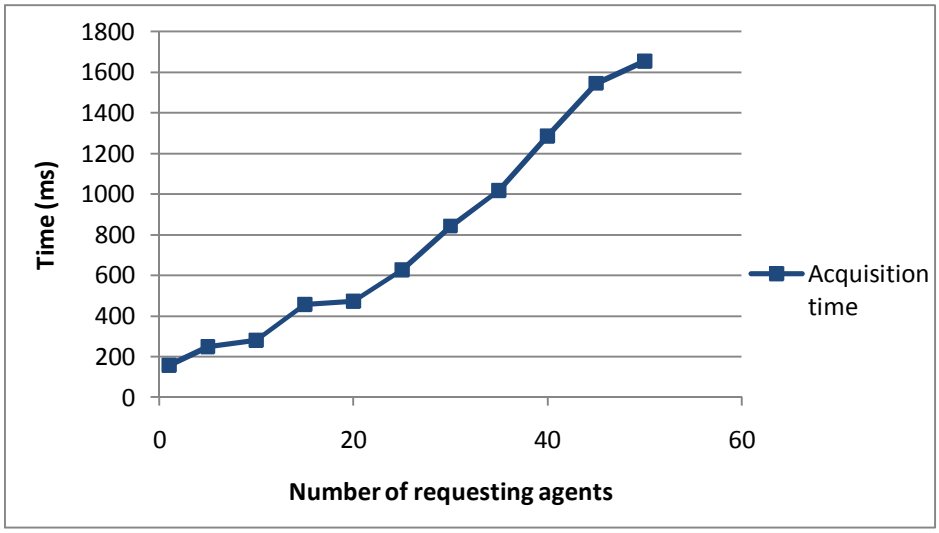


Figure 6-26: Capability acquisition time with increasing number of client agents

The second and third scalability experiments indicate that performance can be adversely affected when an agent has too many alternatives/providers to select from and when a single agent has to serve many client agents. These two cases are representative of decentralized and centralized approaches to capability sharing, and indicate that they are not suited for our approach. We note that the expectation of the peer capability sharing feature is for agents to share capabilities with a few peer agents and not to be mass scale capability sources. However, it is necessary to include necessary safeguards to ensure that an agent does not get flooded with a large number of capability requests and that a capability requesting agent similarly limits its search scope.

While we have not incorporated techniques to avoid such circumstances in our implementation, many possibilities exist, ranging from adding hard limits on number of alternatives/capability requests to consider, to categorising agents based on criteria such as their geography, ownership and load level. In [GLZ09] we proposed one such approach where agents work in teams of workers and capability suppliers. A worker agent concentrates on carrying out user requested goals and depends on a dedicated carrier agent for the necessary capabilities. The carrier agent is responsible for locating suitable capabilities and making them available to the worker.

6.6 Summary

In this chapter we described the experiments which validate the proposed VERSAG framework and its underlying theoretical concepts. A fundamental advantage that a VERSAG agent has over a conventional mobile agent is the ability to acquire new behaviours for situations which are not anticipated at design time. Thus, they are able to continue execution in situations where a conventional mobile agent would have to be replaced with a new one. We described a case study scenario of a personal assistant agent which demonstrated the ability of VERSAG agents to dynamically acquire, shed and share capabilities and the ability to make cost-efficient capability selections. Our evaluations also demonstrated how VERSAG agents bring about performance benefits over conventional mobile agents through the use of their capability sharing feature.

The multi-criteria decision making cost model proposed in chapter 4 is an important part of VERSAG and another contribution of this thesis. We described how the cost model can be applied to make cost-efficient capability acquisition decisions by taking into consideration multiple cost criteria, user specified constraints and contextual conditions. Evaluations presented in this chapter validated the accuracy of the cost model and illustrated how it can be used to improve efficiency of service-oriented mobile software agents in a pervasive computing environment.

In order to maximise its value, the VERSAG framework envisions the existence of a distributed ecosystem of agents. As such, our agents should be able to operate in environments with large numbers of agents. In this chapter we presented experiments which investigate this aspect of VERSAG. The evaluations demonstrated that our agents are linearly scalable when large numbers of agents are co-located. Evaluations on the scalability of the peer capability sharing feature demonstrated that it is not desirable to have an agent consider a large number of alternatives when seeking capabilities. It is therefore necessary to equip VERSAG agents with mechanisms which limit the number of alternatives considered when many are available. The evaluations also illustrated that a single agent should not be required to serve a large number of concurrent capability requests. This is in accordance with our proposal as the expectation is that an agent will share capabilities with a few peers rather than be

a dedicated capability source. It indicates though the necessity of including safeguards to ensure that agents are not inundated with capability requests.

In chapters 3 and 4 we presented our main proposal of VERSAG agents and its underlying concepts. Then, in chapter 5 we described our prototype implementation of VERSAG and presented our evaluations in the current chapter. We now proceed to the concluding chapter of the thesis where the research contributions are summarised and future research directions outlined.

7 Conclusion

In the preceding chapters we have proposed, developed, implemented and validated our versatile self-adaptive agent (VERSAG) framework as an enabling platform for building smart pervasive applications and services. This chapter summarises the thesis and its contributions, and discusses future directions of this work.

7.1 Research Summary

Recent advances in pervasive computing have been led largely by developments in hardware and communication technologies. The development of software systems which can seamlessly make use of these devices and technologies is an active and important research challenge. In this context, mobile agents are a useful and worthy approach for pervasive computing due to their flexibility, scalability and ability to simplify tasks by delegation.

This thesis focused on harnessing and enhancing the potential of mobile agents for building applications and services for pervasive environments. Towards this end, we proposed and developed VERSAG (VErsatile Self-adaptive AGents), a novel platform for building adaptive software agents which combines mobility with compositional adaptation, thus enhancing and revealing the potential for adaptation that mobile agents possess.

In chapter 3 we introduced the VERSAG framework, describing how an agent is built around six primitive operations and the ability to share functional components (termed *capabilities*) with peer agents. We formalised the key concepts surrounding

VERSAG and described the structure of capabilities in depth. In chapter 4, we described and demonstrated how agents achieve greater autonomy by dynamically adapting their constituent capabilities to meet user requirements and environmental conditions. We also proposed a cost model which enables agents to make cost-efficient capability acquisition decisions. This multi-criteria decision making model lets an agent include a diverse range of cost elements ranging from time consumption to monetary costs in its decision making process. In chapter 5 we described our prototype implementation of the VERSAG agent framework. Experimental evaluations conducted using this implementation to verify the concepts proposed in chapters 3 and 4 were presented in chapter 6. In our evaluations we used time and network load, which are key quality metrics in mobile/distributed applications, as the relevant performance criteria. This chapter concludes the thesis by presenting a summary of the research contributions in section 7.2 and outlining future research directions in section 7.3.

7.2 Research Contributions

This section presents the research contributions made in this thesis. This thesis contributes towards building mobile agent based cost-efficient and adaptive applications and services suitable for pervasive computing environments. The specific contributions made by this thesis are as follows:

- *Development of the versatile self-adaptive agent (VERSAG) framework.* This research proposed, designed and implemented a component based framework and agent architecture for building compositionally adaptive mobile software agents. An agent in the proposed VERSAG framework is modelled as an active and itinerary driven mobile entity. This basis makes an agent lightweight and suitable for execution in resource restricted environments. Application-specific behaviours are implemented through reusable software components termed *capabilities* which an agent can acquire and discard at runtime. In addition to dynamically acquiring and discarding new application-specific functionality, an agent is capable of increasing its autonomy through acquisition of new capabilities which improve its core functions such as reasoning ability, context-awareness and interaction protocols. Thus, the

agents are highly versatile, able to transform from being lightweight itinerant agents to more autonomous agents with complex reasoning capacity. This feature is highly desirable in pervasive computing scenarios where the environment as well as application requirements change rapidly. The functional and performance benefits of this proposal have been validated through a prototype implementation and a set of experimental evaluations. Furthermore, in chapter 2 we identified lack of support for component reusability as a drawback of current compositionally adaptive agent systems. Through the use of OSGi in our implementation we were able to create a capability model that adheres to a widely accepted industry standard and hence has a wider scope of reuse.

- *Development of peer capability sharing amongst mobile agents.* The concept of sharing functional components among peer agents was proposed and implemented as a mechanism to increase efficiency and utility of mobile agents in pervasive environments. Previous work in compositionally adaptive mobile software agents typically depended on a centralized source for new components. This can be a disadvantage, especially in pervasive computing environments where dynamic variations in the environment can disrupt communication with the component source (e.g. due to network connectivity issues and possible source failure). High costs involved may also make communicating with a centralized component source undesirable. The ability of agents to request and acquire capabilities from multiple sources including their peer agents is highly valuable in such situations. Peer capability sharing also brings about a significant benefit in the form of efficient agent migration. While network bandwidth reduction is a key benefit attributed to mobile agents, simplistic and static migration strategies employed by most mobile agent toolkits result in poor bandwidth usage. Our proposed peer capability sharing introduces a new layer of code mobility to VERSAG agents, and it is possible for agents to control and implement efficient migration mechanisms at this fine-grained level. The feasibility and benefits of this feature have been validated through our experimental evaluations presented in chapter 6.

- *Development of a multi-criteria decision making cost model for cost-efficient agent adaptation.* A VERSAG agent's itinerant behaviour is constrained by different cost parameters that need to be adhered to. This research proposed and implemented a cost model that allows agents to make cost-efficient adaptation decisions taking into consideration multiple cost criteria, user specified constraints and contextual conditions. While the cost model is not limited to a specific set of cost criteria, time consumption, network usage, computational resource consumption and accuracy of results were used as representative cost criteria. The cost model utilizes *a priori* estimates of different cost elements in order to make decisions on capability acquisition. The cost model and formulae for cost estimates were presented in chapter 4 and its accuracy validated through empirical evaluations in chapter 6.

The results of this thesis have been validated and published in seven peer-reviewed international conference papers [GKL09, GKL10, GLZ09, GZK08a, GZK08b, GZK09, GZK10] and one journal article [GLZ11]. Having highlighted the principal contributions of this thesis, in the next section we briefly outline future research directions of this work.

7.3 Research Directions

This section concludes the thesis by outlining possible future research directions in which our work could be extended.

- *Multiple agent platform support.* The VERSAG agent framework is developed with the intention of supporting multiple agent platforms and allowing cross-platform migration of agents. Since our prototype implementation is limited to the JADE agent platform, a natural future progression is to extend framework 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.

- *Capability description and reasoning mechanisms.* As identified in chapter 3, VERSAG capabilities need to be described to support automated reasoning and matching on them. However, in this research we did not develop specific mechanisms for capability description, and stated that our expectation is for VERSAG to leverage existing research in related fields for this purpose. Therefore, this is a possible direction to develop the current research in.
- *Itinerary generation.* A related future direction for investigation is generating detailed agent itineraries from high-level declarative user requests. Itinerary generation has many issues to be investigated such as decomposing the requested task into a collection of activities, identifying a sequence of locations to visit and selecting a suitable capability management policy. Here, since existing interface specification techniques are unable to ensure component compatibility at behavioural and semantic levels, assistance in the form of pre-generated task decompositions may be necessary.
- *Standards for capability testing.* We also presented the need of specifying standard test environments and a standard body such that capabilities developed by different parties could be tested and annotated with comparable performance parameters (e.g. resource requirements, accuracy of generated results, reliability and speed of execution). This is a future research direction that will improve the utility of VERSAG and make it attractive for third-party capability developers.
- *Expand cost model usage.* The adaptation decision making cost model developed within this thesis has been used only for decision making during capability acquisition. As mentioned in chapter 4, the cost model could be extended for use at other decision making steps such as agent migration, responding to contextual changes and itinerary generation.

With regards to our prototype implementation and evaluations, we identify the following three research directions.

- *Evaluations with more cost elements.* Our implementation and evaluations focused on time consumption and network load as relevant cost elements. Extending these with evaluations to involve resource consumption, accuracy as well as other cost elements is planned for future work.
- *Improved capability sharing performance.* The scalability evaluations of chapter 6 showed that the current implementation of capability sharing can be a performance bottleneck when an agent is faced with a large number of capability requests from peers or has too many capabilities to select from. Therefore it is necessary, as discussed therein, to include necessary safeguards in order to remove this bottleneck.
- *Upgrade OSGi support.* The prototype framework implementation uses *Concierge*, which is an open source implementation of OSGi specification release 3. The current release of OSGi specifications is 4.3 and modifying the prototype to support the latest release is planned for future work.

In conclusion, this thesis takes a step forward and opens up new opportunities in realising the potential of mobile software agents enhanced with compositional adaptation for pervasive computing.

References

[AGH05] Arnold, K., Gosling, J. and Holmes, D., (2005), *"The Java™ Programming Language"*, 4th Edition, Addison-Wesley Professional.

[AKK03] Alonso, E., Kudenko, D. and Kazakov, D., (2003), *"Preface"*, Adaptive Agents and Multi-Agent Systems: Adaptation and Multi-Agent Learning, Lecture Notes in Computer Science, 2636, Springer-Verlag, pp. vi-x.

[Ama06] Amara-Hachmi, N., (2006), *"An Ontology-based Model for Mobile Agents Adaptation in Pervasive Environments"*, Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2006), Dubai/Sharjah, UAE, March 8-11, IEEE Press, pp. 1106-1109.

[AmF05] Amara-Hachmi, N. and Fallah-Seghrouchni, A. E., (2005), *"Towards a Generic Architecture for Self-Adaptive Mobile Agents"*, Proceedings of the European Workshop on Adaptive Agents and Multi-Agent Systems, Paris, France, March 21-22.

[AmF09] Amor, M. and Fuentes, L., (2009), *"Malaca: A Component and Aspect-Oriented Agent Architecture"*, Information and Software Technology, Vol. 51, No. 6, (June), pp. 1052-1065.

[ASF10] Alberola, J. M., Such, J. M., Garcia-Fornes, A., Espinosa, A. and Botti, V., (2010), *"A Performance Evaluation of Three Multiagent Platforms"*, Artificial Intelligence Review, Vol. 34, No. 2, (Aug), pp. 145-176.

[BBL76] Boehm, B. W., Brown, J. R. and Lipow, M., (1976), *"Quantitative Evaluation of Software Quality"*, Proceedings of the 2nd International Conference on Software Engineering (ICSE '76), San Francisco, California, 13-15 October, IEEE Computer Society, pp. 592-605.

[BCG07] Bellifemine, F. L., Caire, G. and Greenwood, D., (2007), *"Developing Multi-agent Systems with JADE"*, John Wiley and Sons, Chichester, UK.

[BCP03] Bellifemine, F., Caire, G., Poggi, A. and Rimassa, G., (2003), *"JADE - A White Paper"*, TILAB Journal "EXP - in search of innovation" Special issue on JADE, Vol. 3, (September), pp. 6-19.

[BCT07] Bellifemine, F., Caire, G., Trucco, T., Rimassa, G. and Mungenast, R., (2007), "*JADE Administrator's Guide"*, Available online: http://jade.tilab.com/doc/administratorsguide.pdf (*accessed August 2010*).

[BFH03] Berman, F., Fox, G. and Hey, A. J. G., (2003), *"Grid Computing: Making the Global Infrastructure a Reality"*, John Wiley and Sons, Chichester, UK.

[BGS01] Beigl, M., Gellersen, H.-W. and Schmidt, A., (2001), *"Mediacups: Experience with Design and Use of Computer-Augmented Everyday Artefacts"*, Computer Networks, Vol. 35, No. 4, pp. 401-409.

[BHM04] Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. and Orchard, D., (2004), "*Web Services Architecture: W3C Working Group Note"*, W3C, Available online: http://www.w3.org/TR/ws-arch/ (*accessed April 2011*).

[BJT02] Brazier, F. M. T., Jonker, C. M. and Treur, J., (2002), *"Principles of Component-Based Design of Intelligent Agents"*, Data Knowledge Engineering, Vol. 41, No. 1, pp. 1-27.

[BKR99] Bredin, J., Kotz, D. and Rus, D., (1999), *"Mobile-Agent Planning in a Market-Oriented Environment"*,Technical Report PCS-TR99-345, Department of Computer Science, Dartmouth College. Available online: http://129.170.213.101/reports/TR99-345-rev0.pdf (*accessed August 2011*).

[BMK05a] Braun, P., Mueller, I., Kowalczyk, R. and Kern, S., (2005), *"Increasing the Migration Efficiency of Java-based Mobile Agents"*, Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, (eds) A. Skowron, J.-P. A. Barthès, L. C. Jain, R. Sun, P. Morizet-Mahoudeaux, J. Liu and N. Zhong, Compiegne, France, 19-22 Sept, IEEE Computer Society, pp. 508 - 511.

[BMS05] Braun, P., Müller, I., Schlegel, T., Kern, S., Schau, V. and Rossak, W., (2005), *"Tracy: An Extensible Plugin-Oriented Software Architecture for*

Mobile Agent Toolkits", Software Agent-Based Applications, Platforms and Development Kits, Whitestein Series in Software Agent Technologies, Birkhäuser Basel, pp. 357-381.

[BoD04] Bourque, P. and Dupuis, R., (2004), *"Guide to the Software Engineering Body of Knowledge"*, 2004 Version, IEEE Computer Society, Los Alamitos, California.

[Bor02] Borselius, N., (2002), *"Mobile Agent Security"*, Electronics & Communication Engineering Journal Vol. 14, No. 5, pp. 211-218.

[BOS02] Brazier, F. M. T., Overeinder, B. J., Steen, M. v. and Wijngaards, N. J. E., (2002), *"Agent Factory: Generative Migration of Mobile Agents in Heterogeneous Environments"*, Proceedings of the 2002 ACM Symposium on Applied Computing, Madrid, Spain, March 10-14, ACM Press, pp. 101 - 106.

[Bra01] Brandt, R., (2001), *"Dynamic Adaptation of Mobile Code in Heterogeneous Environments"*, Diplomarbeit, Technical University of Munich, Munich.

[BrR01] Brandt, R. and Reiser, H., (2001), *"Dynamic Adaptation of Mobile Agents in Heterogeneous Environments"*, Mobile Agents: 5th International Conference, MA 2001 Atlanta, GA, USA, December 2–4, 2001 Proceedings, Lecture Notes in Computer Science, 2240/2001, Springer-Verlag, pp. 70-87.

[BrR04] Braun, P. and Rossak, W., (2004), *"Mobile Agents Basic Concepts, Mobility Models, and the Tracy Toolkit"*, Morgan Kaufmann Publishers.

[BTK05] Braun, P., Trinh, D. and Kowalczyk, R., (2005), *"Integrating a New Mobility Service into the Jade Agent Toolkit "*, Mobility Aware Technologies and Applications: Second International Workshop, MATA 2005, Montreal, Canada, October 17-19, 2005. Proceedings, Lecture Notes in Computer Science, 3744/2005, Springer-Verlag, pp. 354-363.

[BYK01] Baek, J.-W., Yeo, J.-H., Kim, G.-T. and Yeom, H.-Y., (2001), *"Cost Effective Mobile Agent Planning for Distributed Information Retrieval"*, Proceedings of the 21st International Conference on Distributed Computing

Systems (ICDCS 2001), Phoenix, Arizona , USA, 16-19 April, IEEE Computer Society, pp. 65-72

[CaK02] Cardoso, R. S. and Kon, F., (2002), *"Mobile Agents: A Key for Effective Pervasive Computing"*, Proceedings of the ACM OOPSLA 2002 Workshop on Pervasive Computing, Seattle, Washington, USA, 4 November, Available online: http://gsd.ime.usp.br/publications/oopsla2002.pdf (*accessed August 2011).*

[CCB09] Choi, S., Choo, H., Baik, M., Kim, H. and Byun, E., (2009), *"ODDUGI: Ubiquitous Mobile Agent System"*, Computational Science and Its Applications – ICCSA 2009 International Conference, Seoul, Korea, June 29-July 2, 2009, Proceedings, Part II, Lecture Notes in Computer Science, 5593, Springer-Verlag, pp. 393-407.

[CCD99] Chen, Q., Chundi, P., Dayal, U. and Hsu, M., (1999), *"Dynamic Agents"*, International Journal of Cooperative Information Systems, Vol. 8, No. 2-3, pp. 195-223.

[CFJ03] Chen, H., Finin, T. and Joshi, A., (2003), *"An Ontology for Context-Aware Pervasive Computing Environments"*, Knowledge Engineering Review, Vol. 18, No. 3, (Sep), pp. 197-207.

[CGG03] Capera, D., Georgé, J.-P., Gleizes, M.-P. and Glize, P., (2003), *"The AMAS Theory for Complex Problem Solving Based on Self-Organizing Cooperative Agents"*, Proceedings of the Twelfth International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03), Linz, Austria, 9-11 June, IEEE Computer Society, pp. 383-388.

[CGK05] Chmiel, K., Gawinecki, M., Kaczmarek, P., Szymczak, M. and Paprzycki, M., (2005), *"Efficiency of JADE Agent Platform"*, Scientific Programming, Vol. 13, No. 2, pp. 159-172.

[ChK00] Chen, G. and Kotz, D., (2000), *"A Survey of Context-Aware Mobile Computing Research"*,Technical Report TR2000-381, Dartmouth College. Available online: http://www.cs.dartmouth.edu/reports/TR2000-381.pdf (*accessed August 2011*).

[CMP06] Canal, C., Murillo, J. M. and Poizat, P., (2006), *"Software Adaptation"*, L'objet: Special Issue on Coordination and Adaptation Techniques for Software Entities, Vol. 12, No. 1, pp. 9-31.

[CoH01] Councill, B. and Heineman, G. T., (2001), *"Definition of a Software Component and Its Elements"*, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley Longman Publishing, pp. 5-19.

[CPV07] Carzaniga, A., Picco, G. P. and Vigna, G., (2007), *"Is Code Still Moving Around? Looking Back at a Decade of Code Mobility"*, Proceedings of the 29th International Conference on Software Engineering: (ICSE'07 Companion), Minneapolis, MN, USA, 20-26 May, IEEE Computer Society, pp. 9-18.

[Dav02] Davids, A., (2002), *"Urban Search and Rescue Robots: From Tragedy to Technology"*, IEEE Intelligent Systems, Vol. 17, No. 2, pp. 81-83.

[Din02] Dinda, P. A., (2002), *"Online Prediction of the Running Time of Tasks"*, Cluster Computing, Vol. 5, No. 3, pp. 225-236.

[DMP95] Diakoulaki, D., Mavrotas, G. and Papayannakis, L., (1995), *"Determining Objective Weights in Multiple Criteria Problems: The CRITIC Method"*, Computers & Operations Research, Vol. 22, No. 7, pp. 763-770.

[DPK00] Dixon, K. R., Pham, T. Q. and Khosla, P. K., (2000), *"Port-Based Adaptable Agent Architecture"*, Self-Adaptive Software: First International Workshop, IWSAS 2000, Oxford, UK, April 2000. Revised Papers, Lecture Notes in Computer Science, 1936, Springer-Verlag, pp. 181-198.

[DSW05] Das, S., Shuster, K., Wu, C. and Levit, I., (2005), *"Mobile Agents for Distributed and Heterogeneous Information Retrieval"*, Information Retrieval, Vol. 8, No. 3, pp. 383-416.

[DZK09] DelirHaghighi, P., Zaslavsky, A., Krishnaswamy, S. and Gaber, M. M., (2009), *"Mobile Data Mining for Intelligent Healthcare Support"*, Proceedings of the 42nd Hawaii International Conference on System Sciences (HICSS09), Big Island, Hawaii, January 5-8, 2009, IEEE Computer Society Press, pp. 1-10.

[Eck68] Eckenrode, R. T., (1968), *"Weighting Multiple Criteria"*, Management Science, Vol. 12, No. 3, pp. 180-192.

[EdG01] Edwards, W. K. and Grinter, R. E., (2001), *"At Home with Ubiquitous Computing: Seven Challenges"*, Ubicomp 2001: Ubiquitous Computing International Conference Atlanta Georgia, USA, September 30–October 2, 2001 Proceedings, Lecture Notes in Computer Science, 2201/2001, Springer-Verlag, pp. 256-272.

[Edw92] Edwards, S., (1992), *"Towards a Model of Reusable Software Subsystems"*, 5th Annual Workshop On Software Reuse (WISR'92), Palo Alto, California, 28-30 October, Available online: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.9080&rep=rep1&type=pdf, *(accessed August 2011)*.

[EKR08] Erfurth, C., Kern, S., Rossak, W., Braun, P. and Leßmann, A., (2008), *"MobiSoft: Networked Personal Assistants for Mobile Users in Everyday Life"*, Cooperative Information Agents XII 12th International Workshop, CIA 2008, Prague, Czech Republic, September 10-12, 2008. Proceedings, Lecture Notes in Computer Science, 5180, Springer-Verlag, pp. 147-161.

[EmK02] Emmerich, W. and Kaveh, N., (2002), *"Component Technologies: Java Beans, COM, CORBA, RMI, EJB and the CORBA Component Model"*, Proceedings of the 24th International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA, 19-25 May, ACM Press, pp. 691-692.

[Ext99] The Rules of Extreme Programming (1999), Available online: http://www.extremeprogramming.org/rules.html (*accessed August 2011*).

[FCL08] Feng, Y., Cao, J., Lau, I. C. H. and Liu, X., (2008), *"A Self-configuring Personal Agent Platform for Pervasive Computing"*, Proceedings of the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC'08), Shanghai, China, 17-20 December, IEEE Computer Society, pp. 438-444.

[Fip10] The Foundation for Intelligent Physical Agents (2010), Available online: http://www.fipa.org (*accessed February 2011*).

[FlS99] Flinn, J. and Satyanarayanan, M., (1999), *"Energy-Aware Adaptation for Mobile Applications"*, Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99), Kiawah Island, SC, 12-15 December, ACM Press, pp. 48-63.

[FoZ94] Forman, G. H. and Zahorjan, J., (1994), *"The Challenges of Mobile Computing"*, Computer, Vol. 27, No. 4, pp. 38-47.

[GKL09] Gunasekera, K., Krishnaswamy, S., Loke, S. W. and Zaslavsky, A., (2009), *"Runtime Efficiency of Adaptive Mobile Software Agents in Pervasive Computing Environments"*, Proceedings of the ACM International Conference on Pervasive Services (ICPS'09), London, UK, 13-16 July, ACM Press, pp. 123-132.

[GKL10] Gunasekera, K., Krishnaswamy, S., Loke, S. W. and Zaslavsky, A., (2010), *"Adaptation Support for Agent Based Pervasive Systems"*, Proceedings of the 7th International ICST Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous 2010), Sydeny, Australia, 6-9 December.

[GLX05] Guo, R., Le, J. and Xia, X., (2005), *"Capability Matching of Web Services Based on OWL-S"*, Proceedings of the Fifth International Workshop on Web Based Collaboration (WBC 2005) at Sixteenth International Workshop on Database and Expert Systems Applications (DEXA'05), Copenhagen, Denmark, 22-26 August, IEEE Computer Society, pp. 653-657.

[GLZ09] Gunasekera, K., Loke, S. W., Zaslavsky, A. and Krishnaswamy, S., (2009), *"Runtime Adaptation of Multiagent Systems for Ubiquitous Environments"*, Proceedings of the 2009 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT 2009), Milan, Italy, 15-18 September, IEE Computer Society, pp. 486-490.

[GLZ11] Gunasekera, K., Loke, S., Zaslavsky, A. and Krishnaswamy, S., (2011), *"Improving Efficiency of Service Oriented Context-Driven Software Agents"*, Cybernetics and Systems, Vol. 42, No. 5, pp. 324-340.

[Gor06] Gorton, I., (2006), *"Essential Software Architecture"*, Springer-Verlag, Berlin Heidelberg.

[GrS00] Grandison, T. and Sloman, M., (2000), *"A Survey of Trust in Internet Applications"*, IEEE Communications Surveys, Vol. 3, No. 4, pp. 2-16.

[Gue04] Guessoum, Z., (2004), *" Adaptive Agents and Multiagent Systems"*, IEEE Distributed Systems Online, Vol. 5, No. 7.

[GZK08a] Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2008), *"Context Driven Compositional Adaptation of Mobile Agents"*, Proceedings of the International Workshop on Data Management in Context-Aware Computing (DMCAC 2008), Beijing, China, 27-30 April, IEEE Computer Society, pp. 201-208.

[GZK08b] Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2008), *"VERSAG: Context-Aware Adaptive Mobile Agents for the Semantic Web"*, Proceedings of the 3rd IEEE International Workshop on Engineering Semantic Agent Systems (ESAS 2008), at COMPSAC 2008, Turku, Finland, 28 July - 1 August, IEEE Computer Society, pp. 521-522.

[GZK09] Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2009), *"Component Based Approach for Composing Adaptive Mobile Agents"*, Agent and Multi-Agent Systems: Technologies and Applications 3rd KES International Symposium, KES-AMSTA 2009, Uppsala, Sweden, June 3-5, 2009. Proceedings, Lecture Notes in Computer Science, 5559, Springer-Verlag, pp. 90-99.

[GZK10] Gunasekera, K., Zaslavsky, A., Krishnaswamy, S. and Loke, S. W., (2010), *"Service Oriented Context-Aware Software Agents for Greater Efficiency"*, Agent and Multi-Agent Systems: Technologies and Applications 4th KES International Symposium, KES-AMSTA 2010, Gdynia, Poland, June 23-25, 2010, Proceedings. Part I, Lecture Notes in Computer Science, 6070, Springer-Verlag, pp. 62-71.

[Hay95] Hayes-Roth, B., (1995), *"An Architecture for Adaptive Intelligent Systems"*, Artificial Intelligence, Vol. 72, No. 1-2, pp. 329-365.

[HKB97] Hohl, F., Klar, P. and Baumann, J., (1997), *"Efficient Code Migration for Modular Mobile Agents"*,Technical Report TR-1997-06, University of Stuttgart, Faculty of Computer Science. Available online: ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-1997-06/TR-1997-06.pdf (*accessed August 2011*).

[HKH09] Hirsch, B., Konnerth, T. and Heßler, A., (2009), *"Merging Agents and Services - the JIAC Agent Platform"*, Multi-Agent Programming: Languages, Tools and Applications, Springer-Verlag, pp. 159-185.

[HRH01] Howden, N., Rönnquist, R., Hodgson, A. and Lucas, A., (2001), *"JACK Intelligent Agents™ – Summary of an Agent Infrastructure"*, Proceedings of the Second International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at Agents 2001, Montreal, Canada, 28 May - 1 June, pp. 251-257.

[HwY81] Hwang, C.-L. and Yoon, K., (1981), *"Multiple Attribute Decision Making : Methods and Applications : A State-of-the-Art Survey"*, Springer-Verlag, Berlin Heidelberg.

[ISO01] ISO/IEC, (2001), "*9126-1, Software Engineering – Product Quality – Part 1: Quality Model"*, International Organization for Standardization, Available online: http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749 (*accessed August 2011*).

[ISO08] ISO/IEC, (2008), "*9075-1: Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)"*, International Organization for Standardization, Available online: http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498 (*accessed August 2011*).

[Jad09] Jade - Java Agent DEvelopment Framework (2009), Available online: http://jade.tilab.com/ (*accessed July 2009*).

[Jan00] Jansen, W. A., (2000), *"Countermeasures for Mobile Agent Security"*, Computer Communications, Vol. 23, No. 17, pp. 1667–1676.

[KBF04] Kern, S., Braun, P., Fensch, C. and Rossak, W., (2004), *"Class Splitting as a Method to Reduce Migration Overhead of Mobile Agents"*, On the Move to Meaningful Internet Systems 2004: CoopIS, DOA, and ODBASE, Lecture Notes in Computer Science, 3291, Springer-Verlag, pp. 1358–1375.

[KeC03] Kephart, J. and Chess, D., (2003), *"The Vision of Autonomic Computing"*, Computer, Vol. 36, No. 1, pp. 41 - 50.

[KiF02] Kindberg, T. and Fox, A., (2002), *"System Software for Ubiquitous Computing"*, Computer, Vol. 1, No. 1, pp. 70-81.

[KiL07] Kim, Y.-S. and Lee, K.-H., (2007), *"A Light-weight Framework for Hosting Web Services on Mobile Devices"*, Proceedings of the Fifth European Conference on Web Services (ECOWS '07), Halle, Germany 26-28 Nov, IEEE Computer Society, pp. 255 - 263

[KLM96] Kaelbling, L. P., Littman, M. L. and Moore, A. W., (1996), *"Reinforcement Learning: A Survey"*, Journal of Artificial Intelligence Research, Vol. 4, pp. 237-285.

[KLZ03] Krishnaswamy, S., Loke, S. W. and Zaslavsky, A., (2003), *"Supporting the Optimisation of Distributed Data Mining by Predicting Application Run Times"*, Enterprise Information Systems IV, Kluwer Academic Publishers, pp. 142-149.

[KyD00] Kvarnström, J. and Doherty, P., (2000), *"TALplanner: A Temporal Logic Based Forward Chaining Planner"*, Annals of Mathematics and Artificial Intelligence, Vol. 30, No. 1-4, pp. 119-169.

[Lad00] Laddaga, R., (2000), *"Active Software"*, Self-Adaptive Software: First International Workshop, IWSAS 2000 Oxford, UK, April 17–19, 2000 Revised Papers, Lecture Notes in Computer Science, 1936, Springer-Verlag, pp. 11-26.

[Lad97] Laddaga, R., (1997), "*Self-Adaptive Software, DARPA, BAA-98-12, Proposer Information Pamphlet"*, Available online: http://people.csail.mit.edu/rladdaga/BAA98-12excerpt.html (*accessed August 2011*).

[LaO98] Lange, D. B. and Oshima, M., (1998), *"Programming and Deploying Java Mobile Agents with Aglets"*, Addison-Wesley Longman Publishing, Boston, MA, USA.

[LaO99] Lange, D. B. and Oshima, M., (1999), *"Seven Good Reasons for Mobile Agents"*, Communications of the ACM, Vol. 42, No. 3, pp. 88-89.

[LaW07] Lau, K.-K. and Wang, Z., (2007), *"Software Component Models"*, IEEE Transactions on Software Engineering, Vol. 33, No. 10, (Oct 7), pp. 709-724.

[LOK97] Lange, D. B., Oshima, M., Karjoth, G. and Kosaka, K., (1997), *"Aglets: Programming Mobile Agents in Java"*, Worldwide Computing and Its Applications: International Conference, WWCA'97 Tsukuba, Japan, March 10–11, 1997 Proceedings, Lecture Notes in Computer Science, 1274, Springer-Verlag, pp. 253-266.

[LSW11] Lamping, U., Sharpe, R. and Warnicke, E., (2011), "*Wireshark User's Guide"*, Available online: www.wireshark.org/download/docs/user-guide-a4.pdf (*accessed August 2010*).

[LuX05] Lu, S. and Xu, C.-Z., (2005), *"A Formal Framework for Agent Itinerary Specification, Security Reasoning and Logic Analysis"*, Proceedings of the Third International Workshop on Mobile Distributed Computing (MDC) (ICDCSW'05), Columbus, Ohio, USA, 6-10 June, IEEE Computer Society, pp. 580-586.

[LZY01] Loke, S. W., Zaslavsky, A., Yap, B. and Fonseka, J. R., (2001), *"An Itinerary Scripting Language for Mobile Agents in Enterprise Applications"*, Proceedings of the 2nd Asia-Pacific Conference on Intelligent Agent Technology (IAT 2001), (eds) N. Zhong, J. Liu, S. Ohsuga and J. Bradshaw, Maebashi, Japan, 23-26 October, World Scientific Publishing, pp. 124-128.

[Mae94] Maes, P., (1994), *"Modeling Adaptive Autonomous Agents"*, Artificial Life, Vol. 1, No. 1-2, pp. 135-162.

[MaK01] Marples, D. and Kriens, P., (2001), *"The Open Services Gateway Initiative: An Introductory Overview"*, IEEE Communications Magazine, Vol. 39, No. 12, pp. 110-114.

[MaM06] Marín, C. A. and Mehandjiev, N., (2006), *"A Classification Framework of Adaptation in Multi-Agent Systems"*, Cooperative Information Agents X: 10th International Workshop, CIA 2006 Edinburgh, UK, September 11-13, 2006 Proceedings, Lecture Notes in Computer Science, 4149, Springer-Verlag, pp. 198-212.

[MBM07] Martin, D., Burstein, M., McDermott, D., McIlraith, S., Paolucci, M., Sycara, K., McGuinness, D. L., Sirin, E. and Srinivasan, N., (2007), *"Bringing Semantics to Web Services with OWL-S"*, World Wide Web, Vol. 10, No. 3, pp. 243-277.

[Mil99] Milojicic, D. S., (1999), *"Trend Wars - Mobile Agent Applications "*, IEEE Concurrency, Vol. 7, No. 3, pp. 80-90.

[MKL02] Milojicic, D. S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S. and Xu, Z., (2002), *"Peer-to-Peer Computing"*,Technical Report HPL-2002-57R.1, HP Laboratories, Palo Alto, USA. Available online: http://www.hpl.hp.com/techreports/2002/HPL-2002-57R1.html (*accessed July 2011*).

[MOB07] Muldoon, C., O'Hare, G. M. P. and Bradley, J. F., (2007), *"Towards Reflective Mobile Agents for Resource-Constrained Mobile Devices"*, Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'07), Honolulu, Hawai, 14-18 May, ACM Press, pp. 932-934.

[Moi98] Moizumi, K., (1998), *"Mobile Agent Planning Problems"*, PhD Thesis, Dartmouth College, Hanover, NH.

[MSA06] Mulet, L., Such, J. M. and Alberola, J. M., (2006), *"Performance Evaluation of Open Source Multiagent Platforms"*, Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), Hakodate, Hokkaido, Japan, 8-12 May, ACM Press, pp. 1107-1109.

[MSK04a] McKinley, P. K., Sadjadi, S. M., Kasten, E. P. and Cheng, B. H. C., (2004), *"A Taxonomy of Compositional Adaptation"*,Technical Report MSU-

CSE-04-17, Department of Computer Science and Engineering, Michigan State University. Available online: ftp://ftp.cse.msu.edu/pub/crg/PAPERS/survey-tr.pdf (*accessed August 2011*).

[MSK04b] McKinley, P. K., Sadjadi, S. M., Kasten, E. P. and Cheng, B. H. C., (2004), *"Composing Adaptive Software"*, Computer, Vol. 37, No. 7, pp. 56-64.

[NCL99] Nau, D., Cao, Y., Lotem, A. and Munoz-Avila, H., (1999), *"SHOP: Simple Hierarchical Ordered Planner"*, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99), Stockholm, Sweden, 31 July - 6 August, Morgan Kaufmann Publishers, pp. 968-973.

[NiL04] Niemelä, E. and Latvakoski, J., (2004), *"Survey of Requirements and Solutions for Ubiquitous Software"*, Proceedings of the 3rd International Conference on Mobile and Ubiquitous Multimedia (MUM2004), College Park, Maryland, 27-29 October, ACM Press, pp. 71-78.

[NRT04] Ni, Q., Romdhani, L. and Turletti, T., (2004), *"A Survey of QoS Enhancements for IEEE 802.11 Wireless LAN"*, Wireless Communications & Mobile Computing, Vol. 4, No. 5, (Aug), pp. 547-566.

[OgO05] O'Grady, M. J. and O'Hare, G. M. P., (2005), *"Mobile Devices and Intelligent Agents: Towards a New Generation of Applications and Services"*, Information Sciences, Vol. 171, No. 4, pp. 335-353.

[OSG03] OSGi Alliance, (2003), *"OSGi Service Platform, Release 3"*, IOS Press, Inc., Amsterdam, The Netherlands.

[OSG07] OSGi Alliance, (2007), *"About the OSGi Service Platform: Technical Whitepaper"*, OSGi Alliance. Available online: www.osgi.org/wiki/uploads/Links/OSGiTechnicalWhitePaper.pdf (*accessed August 2011*).

[PaG03] Papazoglou, M. P. and Georgakopoulos, D., (2003), *"Service-Oriented Computing"*, Communications of the ACM, Vol. 46, No. 10, (Oct), pp. 25-28.

[PaL05] Panait, L. and Luke, S., (2005), *"Cooperative Multi-Agent Learning: The State of the Art"*, Autonomous Agents and Multi-Agent Systems, Vol. 11, No. 3, (November), pp. 387-434.

[PAP04] Paprzycki, M., Abraham, A., Pirvanescu, A., Badica, C. and Bădică, C., (2004), *"Implementing Agents Capable of Dynamic Negotiation"*, Proceedings of the 6th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC04), (eds) P. Petcu and V. Negru, Timisoara, Romania, 26-30 September, Mirton Press, pp. 369–380.

[Pay86] Payton, D., (1986), *"An Architecture For Reflexive Autonomous Vehicle Control"*, Proceedings of the IEEE International Conference on Robotics and Automation, San Fransicso, CA, USA, 7-10 April, IEEE Press, pp. 1838-1845.

[Pic01] Picco, G. P., (2001), *"Mobile Agents: An Introduction"*, Microprocessors and Microsystems, Vol. 25, pp. 65-74.

[Pic98] Picco, G. P., (1998), *"μCode: A Lightweight and Flexible Mobile Code Toolkit"*, Mobile Agents: Second International Workshop, MA'98 Stuttgart, Germany, September 9–11, 1998 Proceedings, Lecture Notes in Computer Science, 1477, Springer-Verlag, pp. 160-171.

[Pit90] Pitrat, J., (1990), *"An Intelligent System Must and Can Observe its Own Behavior"*, Proceedings of the Third COGNITIVA Symposium on At the Crossroads of Artificial Intelligence, Cognitive Science, and Neuroscience, (eds) T. Kohonen and F. Fogelman-Soulie, Madrid, Spain, 20-23 November, North-Holland Publishing Co, pp. 119-128.

[PKA07] Price, R., Krishnaswamy, S. and Arora, N., (2007), *"Current Research in Conceptual Modelling of Agent Mobility: An Ontology-Based Evaluation"*, International Journal of Metadata, Semantics and Ontologies, Vol. 2, No. 2, pp. 79 - 93.

[PPN02] Parakh, G., Paprzycki, M. and Nistor, C. E., (2002), *"Dynamically Loaded Reasoning Models in Negotiating Agents"*, Proceedings of the 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, 26-27 September, pp. 199-203.

[PrB08a] Preuveneers, D. and Berbers, Y., (2008), *"Pervasive Services on the Move: Smart Service Diffusion on the OSGi Framework"*, Ubiquitous Intelligence and Computing: 5th International Conference, UIC 2008, Oslo, Norway, June 23-25, 2008 Proceedings, Lecture Notes in Computer Science, 5061, Springer-Verlag, pp. 46-60.

[PrB08b] Preuveneers, D. and Berbers, Y., (2008), *"Encoding Semantic Awareness in Resource-Constrained Devices"*, IEEE Intelligent Systems, Vol. 23, No. 2, (Mar-Apr), pp. 26-33.

[Pre01] Pressman, R. S., (2001), *"Component-Based Software Engineering"*, Software Engineering: A Practitioner's Approach, McGraw-Hill Series in Computer Science, McGraw-Hill, pp. 721-746.

[PTD08] Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F., (2008), *"Service-Oriented Computing: A Research Roadmap"*, Cooperative Information Systems, Vol. 17, No. 2, (June), pp. 223-255.

[PZI04] Page, J., Zaslavsky, A. and Indrawan, M., (2004), *"A Buddy Model of Security for Mobile Agent Communities Operating in Pervasive Scenarios"*, Proceedings of the Second Australasian Information Security Workshop (AISW 2004), (eds) J. Hogan, P. Montague, M. Purvis and C. Steketee, Dunedin, New Zealand, Australian Computer Society, pp. 17-25.

[QiW01] Qi, H. and Wang, F., (2001), *"Optimal Itinerary Analysis for Mobile Agents in Adhoc Wireless Sensor Networks"*, Proceedings of the 13th International Conference on Wireless Communications, Calgary, Canada, July 9-11, pp. 147-152.

[RaC03] Ranganathan, A. and Campbell, R. H., (2003), *"A Middleware for Context-Aware Agents in Ubiquitous Computing Environments"*, Middleware 2003: ACM/IFIP/USENIX International Middleware Conference Rio de Janeiro,

Brazil, June 16–20, 2003 Proceedings, Lecture Notes in Computer Science, 2672, Springer-Verlag, pp. 143-161.

[RaG95] Rao, A. S. and Georgeff, M. P., (1995), *"BDI Agents: From Theory to Practice"*, Proceedings of the First International Conference on Multiagent Systems (ICMAS95), San Francisco, CA, USA, 12-14 June, MIT Press, pp. 312-319.

[RCC04] Ranganathan, A., Chetan, S. and Campbell, R. H., (2004), *"Mobile Polymorphic Applications in Ubiquitous Computing Environments"*, Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous 2004), Networking and Services, Cambridge, USA, 22-26 August, IEEE Computer Society, pp. 402-411.

[ReA07] Rellermeyer, J. S. and Alonso, G., (2007), *"Concierge: A Service Platform for Resource-Constrained Devices"*, Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Portugal, 21-23 March, ACM Press, pp. 245-258.

[RiR07] Richardson, L. and Ruby, S., (2007), *"RESTful Web Services"*, First Edition, O'Reilly Media, Sebastopol, CA, USA.

[RKL05] Roman, D., Keller, U., Lausen, H., Bruijn, J. d., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C. and Fensel, D., (2005), *"Web Service Modeling Ontology"*, Applied Ontology, Vol. 1, No. 1, pp. 77-106.

[Saa08] Saaty, T. L., (2008), *"Decision Making with the Analytic Hierarchy Process"*, International Journal of Services Sciences, Vol. 1, No. 1, pp. 83-98.

[Saa77] Saaty, T. L., (1977), *"Scaling Method for Priorities in Hierarchical Structures"*, Journal of Mathematical Psychology, Vol. 15, No. 3, pp. 234-281.

[Saa90] Saaty, T. L., (1990), *"How to Make a Decision - the Analytic Hierarchy Process"*, European Journal of Operational Research, Vol. 48, No. 1, (Sep), pp. 9-26.

[SaM03] Saha, D. and Mukherjee, A., (2003), *"Pervasive Computing: A Paradigm for the 21st Century"*, Computer, Vol. 36, No. 3, pp. 25-31.

[SAM07] Such, J. M., Alberola, J. M., Mulet, L., Espinosa, A., Garcia-Fornes, A. and Botti, V., (2007), *"Large-Scale Multiagent Platform Benchmarks"*, First International Workshop on Languages, Methodologies and Development Tools for Multi-Agent Systems (LADS 2007), Durham, UK, September 4-6, 2007, Available online: http://lia.deis.unibo.it/confs/lads/papers/5.1%20paper_13%20(such).pdf *(accessed August 2011)*.

[Sat01] Satyanarayan, M., (2001), *"Pervasive Computing: Vision and Challenges"*, IEEE Personal Communications, Vol. 8, No. 4, pp. 10-17.

[SaT09] Salehie, M. and Tahvildari, L., (2009), *"Self-Adaptive Software: Landscape and Research Challenges"*, ACM Transactions on Autonomous and Adaptive Systems (TAAS), Vol. 4, No. 2, pp. 14:01-42.

[Sat10] Satoh, I., (2010), *"Mobile Agent-based Context-aware Services"*, Journal of Universal Computer Science, Vol. 16, No. 15, pp. 1929-1952.

[SAW94] Schilit, B., Adams, N. and Want, R., (1994), *"Context-Aware Computing Applications"*, Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications, Santa Cruz, California, 8-9 December, IEEE Computer Society, pp. 85-90.

[SBB00] Suri, N., Bradshaw, J. M., Breedy, M. R., Groth, P. T., Hill, G. A. and Jeffers, R., (2000), *"Strong Mobility and Fine-Grained Resource Control in NOMADS"*, Agent Systems, Mobile Agents, and Applications: Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000, Zurich, Switzerland, September 13-15, 2000 Proceedings, Lecture Notes in Computer Science, 1882, Springer-Verlag, pp. 2-15.

[SFT98] Smith, W., Foster, I. and Taylor, V., (1998), *"Predicting Application Run Times Using Historical Information "*, Job Scheduling Strategies for Parallel Processing: IPPS/SPDP'98 Workshop Orlando, Florida, USA, March 30, 1998 Proceedings, LNCS, 1459/1998, Springer-Verlag, pp. 122-142.

[SGB10] Stahl, F., Gaber, M. M., Bramer, M. and Yu, P. S., (2010), *"Pocket Data Mining: Towards Collaborative Data Mining in Mobile Computing Environments"*, Proceedings of the 22nd International Conference on Tools with Artificial Intelligence, Arras, France, 27-29 October, IEEE Computer Society, pp. 323-330.

[SKG09] Steller, L., Krishnaswamy, S. and Gaber, M., (2009), *"Enabling Scalable Semantic Reasoning for Mobile Services"*, International Journal on Semantic Web and Information Systems, Vol. 5, No. 2, pp. 91-116.

[SLK98] Sycara, K., Lu, J. and Klusch, M., (1998), *"Interoperability among Heterogeneous Software Agents on the Internet"*,Technical Report CMU-RI-TR-98-22, The Robotics Institute, Carnegie Mellon University. Available online: http://www.daml.org/services/owl-s/pub-archive/CMU-RI-TR-98-22.pdf (*accessed August 2011*).

[Som11] Sommerville, I., (2011), *"Component-Based Software Engineering"*, Software Engineering, Addison-Wesley, pp. 452-478.

[SRD09] Shimizu, S., Rangaswami, R., Duran-Limon, H. A. and Corona-Perez, M., (2009), *"Platform-Independent Modeling and Prediction of Application Resource Usage Characteristics"*, Journal of Systems and Software, Vol. 82, No. 12, (Dec), pp. 2117-2127.

[Sse10] Ssekibuulea, R., (2010), *"Mobile Agent Security Against Malicious Platforms"*, Cybernetics and Systems, Vol. 41, No. 7, pp. 522-534.

[SSP04] Spyrou, C., Samaras, G., Pitoura, E. and Evripidou, P., (2004), *"Mobile Agents for Wireless Computing: The Convergence of Wireless Computational Models with Mobile-Agent Technologies"*, Mobile Networks & Applications, Vol. 9, No. 5, (Oct), pp. 517-528.

[StW05] Steinmetz, R. and Wehrle, K., (2005), *"What is This "Peer-to-Peer" About?"*, Peer-to-Peer Systems and Applications, Lecture Notes in Computer Science, 3485, Springer-Verlag, pp. 9-16.

[SWB03] Splunter, S. v., Wijngaards, N. J. E. and Brazier, F. M. T., (2003), *"Structuring Agents for Adaptation"*, Adaptive Agents and Multi-Agent

Systems, Lecture Notes in Computer Science, 2636, Springer-Verlag, pp. 174-186.

[SWB04] Splunter, S. v., Wijngaards, N. J. E., Brazier, F. M. T. and Richards, D., (2004), *"Automated Component-Based Configuration: Promises and Fallacies"*, Proceedings of the 4th Symposium on Adaptive Agents and Multi-Agent Systems at AISB 2004, Leeds, UK, 29-30 March, pp. 130-135.

[SWK02] Sycara, K., Widoff, S., Klusch, M. and Lu, J., (2002), *"Larks: Dynamic Matchmaking Among Heterogeneous Software Agents in Cyberspace"*, Autonomous Agents and Multi-Agent Systems, Vol. 5, No. 2, pp. 173-203.

[TeM01] Tewari, G. and Maes, P., (2001), *"A Generalized Platform for the Specification, Valuation, and Brokering of Heterogeneous Resources in Electronic Markets"*, E-Commerce Agents: Marketplace Solutions, Security Issues, and Supply and Demand, Lecture Notes in Computer Science, 2033, Springer-Verlag, pp. 7-24.

[TGM98] Tu, M. T., Griffel, F., Merz, M. and Lamersdorf, W., (1998), *"A Plug-in Architecture Providing Dynamic Negotiation Capabilities for Mobile Agents"*, Mobile Agents: Second International Workshop, MA'98 Stuttgart, Germany, September 9–11, 1998 Proceedings, Lecture Notes in Computer Science, 1477, Springer-Verlag, pp. 222-236.

[Tra90] Tracz, W., (1990), *"Where Does Reuse Start?"*, ACM SIGSOFT Software Engineering Notes, Vol. 15, No. 2, pp. 42-46.

[Tri00] Triantaphyllou, E., (2000), *"Multi-Criteria Decision Making Methods: A Comparative Study"*, Kluwer Academic Publishers, Dordrecht, The Netherlands.

[TrM89] Triantaphyllou, E. and Mannb, S. H., (1989), *"An Examination of the Effectiveness of Multi-Dimensional Decision-Making Methods: A Decision-Making Paradox"*, Decision Support Systems, Vol. 5, No. 3, pp. 303-312.

[Try07] Tryllian ADK (2007), Available online: http://www.tryllian.org/ (*accessed November 2007*).

[TSG01] Tu, M. T., Seebode, C., Griffel, F. and Lamersdorf, W., (2001), *"DynamiCS: An Actor-Based Framework for Negotiating Mobile Agents"*, Electronic Commerce Research, Vol. 1, No. 1-2, pp. 101-117.

[TsR03] Tsoumakos, D. and Roussopoulos, N., (2003), *"A Comparison of Peer to Peer Search Methods"*, Proceedings of the International Workshop on the Web and Databases (WebDB), (eds) V. Christophides and J. Freire, San Diego, California, 12-13 June, pp. 61-66.

[Ver01] Vercouter, L., (2001), *"A Distributed Approach to Design Open Multi-agent Systems"*, Engineering Societies in the Agents World II: Second International Workshop, ESAW 2001 Prague, Czech Republic, July 7, 2001 Revised Papers, Lecture Notes in Computer Science, 2203, Springer-Verlag, pp. 25-38.

[Vig98] Vigna, G., (1998), *"Mobile Code Technologies, Paradigms, and Applications"*, PhD Thesis, Politecnico di Milano, Milano, Italy.

[Voy07] Voyager Edge (2007), Available online: http://www.recursionsw.com/Products/voyager.html (*accessed November 2007*).

[Wei91] Weiser, M., (1991), *"The Computer for the 21st Century"*, Scientific American, (September).

[Wei96] Weiß, G., (1996), *"Adaptation and Learning in Multi-Agent Systems: Some Remarks and a Bibliography"*, Adaption and Learning in Multi-Agent Systems: IJCAI'95 Workshop Montréal, Canada, August 21, 1995 Proceedings, Lecture Notes in Computer Science, 1042, Springer-Verlag, pp. 1-21.

[WeS01] Weinreich, R. and Sametinger, J., (2001), *"Component Models and Component Services: Concepts and Principles"*, Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley Longman Publishing, pp. 33-48.

[Whi95] Whittle, B., (1995), *"Models and Languages for Component Description and Reuse"*, ACM SIGSOFT Software Engineering Notes, Vol. 20, No. 2, pp. 76-89.

[Woo00] Wooldridge, M., (2000), *"Reasoning about Rational Agents"*, MIT Press.

[Woo02] Wooldridge, M., (2002), *"An Introduction to Multiagent Systems"*, 1st Edition, John Wiley & Sons Ltd, New York, USA.

[YCW06] Yu, P., Cao, J., Wen, W. and Lu, J., (2006), *"Mobile Agent Enabled Application Mobility for Pervasive Computing"*, Ubiquitous Intelligence and Computing: Third International Conference, UIC 2006, Wuhan, China, September 3-6, 2006. Proceedings, Lecture Notes in Computer Science, 4159, Springer-Verlag, pp. 648-657.

[YLY03] Yang, B., Liu, D.-Y., Yang, K. and Wang, S.-S., (2003), *"Strategically Migrating Agents in Itinerary Graph"*, Proceedings of the 2003 International Conference on Machine Learning and Cybernetics, Xi'an, China, 2-5 November, IEEE Computer Society, pp. 1871-1876.

[Zas04] Zaslavsky, A., (2004), *" Mobile Agents: Can They Assist with Context Awareness?"*, Proceedings of the IEEE International Conference on Mobile Data Management (MDM 2004), Berkeley, California, 19-22 January, IEEE Computer Society, pp. 304- 305.

[ZMN05] Zhu, F., Mutka, M. W. and Ni, L., (2005), *"Service Discovery in Pervasive Computing Environments"*, IEEE Pervasive Computing, Vol. 4, No. 4, (Oct-Dec), pp. 81-90.

[ZMW06] Zhao, J., Mao, X. and Wang, J., (2006), *"Developing Multi-Agent Systems with Dynamic Binding Mechanism"*, Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Hong Kong, 18-22 December, IEEE Computer Society, pp. 52-58.

Appendix A: Acronyms and Glossary

3C model	Identifies what an ideal description of a reusable software component should include (*concept*, *content* and *context*).
3G	Third Generation mobile telecommunication
ACL	Agent Communication Language
AHP	Analytic Hierarchy Process
AMAS	Adaptive Multi-agent System
API	Application Programming Interface
BDI	Belief-Desire-Intention
CA	The Controller Agent (CA) is an agent developed to enable administration of VERSAG agents through a command line interface.
Capability	It is a reusable software component that encapsulates functional behaviours that can be dynamically attached and detached from VERSAG agents.
CMP	A capability management policy specifies generic rules which enable a VERSAG agent to make decisions on when to acquire and discard capabilities.
Compositional adaptation	It is where a software application adapts by changing its internal algorithms and structural components.
CORBA	Common Object Request Broker Architecture
COTS	Commercial-off-the-shelf
CPU	Central Processing Unit
DAMA	Dynamically Adaptable Mobile Agents
DESIRE	DEsign and Specification of Interacting REasoning components
DynamiCS	Dynamically Configurable Software
EJB	Enterprise Java Beans
ER	Evidential Reasoning
FIPA	Foundation for Intelligent Physical Agents
GAMA	Generic Adaptive Mobile Agent architecture
GPS	Global Positioning System
GSM	Global System for Mobile communication

GUI	Graphical User Interface
HSDPA	High-Speed Downlink Packet Access
HTTP	Hypertext transfer protocol
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IR	Infrared
ITAG	ITinerary AGent language
JADE	Java Agent DEvelopment Framework
JAR	Java ARchive file
JRE	Java Runtime Environment
JVM	Java Virtual Machine
LAN	Local Area Network
LARKS	Language for Advertisement and Request for Knowledge Sharing
LEAP	Lightweight Extensible Agent Platform
MARI	Multi-Attribute Resource Intermediary
MAS	Multi-agent System
MCDM	Multiple Criteria Decision Making
Migration strategy	A migration strategy is the approach used for moving agent code as a mobile agent migrates.
Mobile agent	A mobile agent is a software entity that moves from one node to another in a network and continues execution on its own accord.
OSGi	Open Services Gateway initiative framework
OWL	Web Ontology Language
OWL-S	OWL for Services
PB3A	Port-Based Adaptable Agent Architecture
PBA	Port-Based Agent
PBM	Port-Based Module
PC	Personal Computer
PDA	Personal Digital Assistant
QoS	Quality of Service
RMA	Remote Monitoring Agent
RMC	The Remote Monitoring Console is a graphical tool which displays specially formatted messages to provide a bird's eye view of the

	functioning of a VERSAG multi-agent system.
SCPA	Self-Configuring Personal Agent Platform
SDK	Software Development Kit
SLA	Service Level Agreement
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP	Transmission Control Protocol
UIO	Ubiquitous Intelligent Object
UML	Unified Modeling Language
URL	Uniform Resource Locator
VERSAG	VErsatile Self-adaptive AGents framework
Wiki	A type of web page designed so that its content can be edited by anyone who accesses it, using a simplified markup language.
Wireshark	An open source network protocol analyser tool
WLAN	Wireless Local Area Network
WPM	Weighted Product Model
WSDL	Web Service Description Language
WSM	Weighted Sum Model
WSMO	Web Service Modeling Ontology
XML	Extensible Markup Language
XWiki	A free Wiki software platform.

Appendix B: Implementation Details

Folder structure of JADE runtime with VERSAG libraries

```
loc1
|
|-- bundle
|   |-- 2common.jar
|   |-- 2jpodpdfreader.jar
|   |-- 2matcher.jar
|   |-- 2pdfboxreader.jar
|   |-- 2result.jar
|   |-- 2txtreader.jar
|   |-- ahpcostmodel.jar
|   |-- contextapi.jar
|   |-- contextdaemon.jar
|   |-- dummycostmodel.jar
|   |-- gui.jar
|   |-- httpclient.jar
|   |-- icommon.jar
|   |-- insertion.jar
|   |-- iprovider.jar
|   |-- irequester.jar
|   |-- iselector.jar
|   |-- pagui.jar
|   |-- resultgui.jar
|   |-- shell.jar
|   |-- vnclient.jar
|   |-- wscostmodel.jar
|   `-- xwiki.jar
|-- conf
|   |-- agent1.properties
|   |-- agent2.properties
|   |-- agent3.properties
|   |-- ann.properties
|   |-- bob.properties
|   |-- dave.properties
|   |-- jadelog.properties
|   `-- tim.properties
|-- jade.log.0
|-- lib
|   |-- JadeLeap.jar
|   |-- concierge-1.0.0.RC3.jar
|   |-- logformatter.jar
|   `-- versag.jar
`-- loc1.sh
```

Script to start JADE main-container with VERSAG support

```
#/bin/bash

VERSAG_HOME=.
LIB=$VERSAG_HOME/lib

# add libraries to classpath
CLASSPATH=$LIB/JadeLeap.jar:$LIB/concierge-
1.0.0.RC3.jar:$LIB/versag.jar:$VERSAG_HOME/conf

java -cp $CLASSPATH -Djava.util.logging.config.file=conf/jadelog.properties jade.Boot -gui
-container-name loc1 -name VPlatform
```

Script to start JADE container with the CA

```
#/bin/bash

VERSAG_HOME=.
LIB=$VERSAG_HOME/lib

# add libraries to classpath
CLASSPATH=$LIB/JadeLeap.jar:$LIB/tools.jar:$VERSAG_HOME/conf:

java -cp $CLASSPATH jade.Boot -container -container-name loc0
Controller:tools.ca.ControllerAgent
```

Command to start RMC

```
> java -cp ./lib/tools.jar tools.rmc.Main
```

Controller Agent script file

The script file shown below is from the case study scenario of agent Bob. The steps given in the file are as follows:

- Create a new agent named "DAVE" at container loc1. Read configuration data from the file "dave.properties".
- Pause for 5 seconds
- Send an itinerary to agent DAVE as an ACL message. The itinerary instructs DAVE to move to container "DavePC" and start listening for peer capability requests and to start a context server (required for measuring network parameters).
- Then, two other agents "ANN" and "TIM" are created and given similar instructions.

```
# start neighbouring agents for case study
#
create DAVE mma2.jade.JadeAgent loc1 dave.properties
pause 5
to:DAVE
itinerary
loc1=move DavePC
DavePC=start icommon#start iprovider#start contextdaemon

pause 5
create ANN mma2.jade.JadeAgent loc1 ann.properties
pause 5
to:ANN
itinerary
loc1=move AnnPC
AnnPC=start icommon#start iprovider#start contextdaemon

pause 5
create TIM mma2.jade.JadeAgent loc1 tim.properties
pause 5
to:TIM
itinerary
loc1=move TimPC
TimPC=start icommon#start iprovider
```