Runtime Efficiency of Adaptive Mobile Software Agents in Pervasive Computing Environments
Kutila Gunasekera¹, Shonali Krishnaswamy¹, Seng Wai Loke², Arkady Zaslavsky³
¹Faculty of Information Technology, Monash University, Australia
²Department of Computer Science & Computer Engineering, La Trobe University, Australia
³Luleå University of Technology, Sweden
{kutila.gunasekera, shonali.krishnaswamy}@infotech.monash.edu.au, S.Loke@latrobe.edu.au, arkady.zaslavsky@ltu.se

ABSTRACT
Developing software that can support heterogeneous hardware platforms, adapt to dynamic highly variable environments and meet rapidly changing requirements is a challenge of pervasive computing. Dynamic adaptive software agents are a promising approach to develop software for such environments. We previously proposed an agent architecture which brings together features such as runtime adaptivity, agent mobility, platform-independence, context-awareness and lightweightness which are desirable in pervasive environments. In this paper, we propose and analyse a cost model of network load for our runtime adaptive mobile software agents. We compare our adaptive agents against standard non-adaptive mobile agents using this cost model. Subsequent experimental results demonstrate that our agents outperform non-adaptive mobile agents. We also propose and discuss further features which make our adaptive agent based software solution more suitable for pervasive computing environments.

Categories and Subject Descriptors
I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence; C.2.4 [Distributed Systems]: Distributed Applications

General Terms
Algorithms, Performance, Experimentation

Keywords
Pervasive computing, mobile agents, adaptive software, cost model

1. INTRODUCTION
The era of disappearing technologies envisioned by Weiser [1] is dawning upon us. Everyday objects with computing capability, miniature sensors and ability to communicate wirelessly with other devices are becoming commonplace. Developing applications and services on top of such infrastructure is a major challenge of pervasive computing [2]. A pervasive application has to execute in a dynamic environment, support heterogeneous devices, use varying methods of communication and adapt to rapidly changing requirements. Many approaches for developing pervasive services have been proposed, and mobile software agents are more and more seen as an attractive option [3, 4]. Flexibility, scalability and ability to reduce complexity by delegation are some of the desirable features that mobile agents bring to pervasive computing applications [5]. Nevertheless, a statically-defined agent is limited in its ability to cope within an uncertain environment, and adaptive agent systems are being pursued as a way of overcoming this limitation [6]. Adaptation can be at multi-agent system (MAS) level or at individual agent level. Agent learning [7, 8] and compositional adaptation are two key approaches in which an individual agent can be made adaptive. While agent learning has been the focus of a considerable amount of research, compositional adaptation of agents has received relatively less attention.

VERSAG is a framework supporting runtime compositional adaptation of mobile agents [9]. It is agent platform independent and is targeted specifically towards pervasive computing environments. Application-specific functionalities in VERSAG are implemented as reusable components, and the framework contains an agent shell that provides an execution environment for these reusable components. The components are agent-platform agnostic and can be reused on heterogeneous agent platforms or other environments where a matching execution environment is available. An agent can adapt by acquiring new components at runtime, thereby gaining completely new behaviours, shedding previous behaviours and changing itself to suit different environments.

The various advantages that mobile agents bring to distributed computing scenarios have been extensively discussed in the literature [10, 11]. It is generally accepted that mobile agents produce less network traffic in comparison to the client-server paradigm in certain situations. Some have suggested that the decision whether to use mobile agents to solve a particular problem should be made at design-time [12] whereas others have recommended the use of a hybrid approach which allows making the decision at runtime [13]. As mentioned above, mobile agents also present other advantages for pervasive computing environments. However, the ability to reduce and control network traffic is an important criterion, especially in wireless networks...
where limited bandwidth and high costs are common. The VERSAG framework contains several mechanisms which allow it to be even more cost-efficient in terms of network usage in comparison to mobile agents. In [14] it was shown how VERSAG can reduce traffic over expensive network links by its adaptive decision making process. In this paper, we progress further in this direction and present a cost model of network load for our runtime adaptive mobile software agents. We compare this against the network cost of standard non-adaptive mobile agents. Experiments conducted to verify the comparison are also described.

The rest of this paper is structured as follows. In section 2, we first go through some previous work on runtime adaptive mobile agents and then describe our approach together with a supporting application scenario. In section 3, the network cost model is explained. Section 4 presents our experimental evaluation and discusses the results obtained. Finally, we conclude in section 5.

2. RUNTIME ADAPTIVE MOBILE AGENTS

2.1 Previous Work

One of the pioneering researches on compositionally adaptive mobile software agents is the Dynamic agent infrastructure [15] which aims to provide agents with dynamic modifiability of behaviours. It is a complete agent architecture built on top of the Java language. Individual Dynamic agents [15] are general purpose carriers of programmes which provide it with intelligence. An agent can dynamically load new programs when it is presented with a task that requires capabilities beyond what it has at present. New components are located either from information available in request messages or by querying a resource-broker. Then a component is downloaded from a URL and added to a local object store. Dynamically Configurable Software (DynamiCS) [16] is another pioneering effort to integrate “mobile” and “intelligent” agent technologies. It provides an architecture which allows dynamic inclusion of negotiating capabilities into mobile agents. Negotiating agents [17] is a project with similar aims, with the main difference being that it is built on top of an existing agent toolkit. The Port-Based Adaptable Agent Architecture (PB3A) [19], Component Architecture for Service Agents (CASA) [20] and the Dutch Agent Factory [21] also contain agents which exhibit compositional adaptivity. An agent-oriented software engineering approach to dynamic agent systems is presented in [22]. This approach, which uses an abstraction named Cast to specify agent behaviours, does not however come with an implementation.

2.2 Versatile Self-Adaptive Agents

VERSAG (VERsatile Self-adaptive AGents) is a novel approach to develop smart pervasive applications through the use of dynamic compositionally adaptive mobile software agents [9]. These software agents are context-driven, adapt by acquiring new software components at runtime, and execute on dynamic heterogeneous environments. VERSAG’s framework allows agents of different architectures to be embedded within it. Also, an agent’s useful functionality is provided in the form of reusable software components.

The salient features of the proposed solution are: the ability of agents to acquire new behaviours from peer agents without depending on designated component providers and an agent’s ability to adapt itself based on contextual input. Self-adaptiveness is important in pervasive applications to manage the growing complexity of applications and to be able to execute in different environments. Agents are carriers of software components; thereby making themselves reusable and easily extensible. Since it does not require any changes to the underlying agent platform, it is possible to use VERSAG agents on already deployed agent networks. The reusable software components in VERSAG are termed as Capabilities.

A VERSAG agent’s high-level task is to execute an itinerary assigned to it. An itinerary is assumed to be similar to the itinerary described in the ITAG language [23] with the main exception that we see an itinerary as belonging to a single agent. It also specifies a list of places the agent has to traverse and actions to execute at each location. An action is the unit of work an agent has to carry out at a particular location. To carry out an action, the agent may need multiple capabilities. The agent decides when and from where the necessary capabilities are acquired and discarded. It may load necessary capabilities in advance, or load them at a later location based on criteria such as capability availability, number of locations a particular capability is required at, network cost and resource constraints at locations. An agent also has the ability to pass on its capabilities to other agents when requested, making all VERSAG agents potential capability providers.

Figure 1. Architecture of a reference VERSAG agent
2.2.1 Reference Architecture

Figure 1 illustrates the architecture of a reference VERSAG agent. The kernel controls the agent and passes control to other modules when required. The capability repository is where the agent stores its application specific capabilities. The itinerary service holds the agent’s itinerary and provides methods to interpret itinerary commands. The capability execution service provides the means to load, run and stop capabilities that are available in the repository. It provides a standard environment on which capabilities can execute independent of the underlying agent platform. The adaptation service contains the logic which allows an agent to adapt. The adaptation process would involve removing and acquiring capabilities, making changes to running capabilities, selecting suitable capabilities from multiple available ones etc. The context service is a related capability which influences the adaptation of an agent. It is expected to consume external context services and also to maintain internal (agent) context information and makes the agent context-aware. The capability exchange service fulfils the dual roles of capability requestor and provider. In its provider role it would listen for capability requests from peer agents and respond as appropriate. The kernel is built on top of an existing mobile agent toolkit (e.g. JADE) and implementations of the base agent for different platforms would allow cross-platform migration of agents. The capabilities themselves are independent of the underlying agent platform as they only access services provided by the VERSAG framework.

2.2.2 Agent Kernel

The agent kernel implements the process coordinator pattern [24] and provides the agent with an itinerary driven behaviour.

A VERSAG agent is thus an itinerary driven agent at its most basic level. More complex behaviours and intelligence are acquired by incorporating appropriate capabilities into the agent. The kernel is able to distinguish and execute a limited set of commands: executing a capability in the repository, searching for a capability, stopping or removing a capability, moving to a different location and terminating the agent. Presented next in pseudo code is the kernel’s execution logic.

```
Get command from Itinerary Service
Check and update status of previous command
If command is EMPTY or previous command is not completed
    Return
If previous command status is FAILED_*,
    Call failure handling procedure
If command is MOVE or TERMINATE
    Stop all running capabilities
    Request MOVE or TERMINATE
    Set command status to EXECUTING and set as previous command
    Return
If command is UNLOAD or STOP
    If specified capability is running, make stop request
    If command is UNLOAD
        Remove capability from repository
        Set command status to EXECUTED and set as previous command
        Return
    If command is FIND
        Get a Requester Service (from Executor Service)
        If Requester Service is unavailable
            Set command status to FAILED_NO_REQUESTER_SERVICE
            Set command as previous command
        Else
            Invoke Requester Service to find capability
            Set command to EXECUTING and set as previous command
        End
End
```

Figure 3. Kernel Execution Process

2.2.3 Capability Model

A capability represents a central concept in VERSAG and is an agent platform-independent software component which can be attached to and detached from a software agent to provide the agent with a particular behaviour.

A capability was formally defined in [9] as a tuple \(<\text{id}, \text{F}, \text{credentials}, \text{Env}>\) where,

- \(\text{id}\) is a unique identifier,
- \(\text{F}\) represents the set of functions that the capability contains,
- \(\text{credentials}\) represent meta-data such as origin, version, security certificates, algorithms and units used and optimizations in the capability and
Capability exchange among peer agents gives VERSAG several advantages over systems where runtime component loading depends on a designated provider. Designated providers need to have high-performance and be highly available to satisfactorily serve requests from a large number of agents, and failure of a provider or broker could cripple the system. For example, in an environment where a group of agents form an ad-hoc network and are out of reach of a designated component provider, an agent which needs to acquire a component is unable to do so even if a neighbouring agent has the component because the provider is unavailable. Peer capability exchange becomes highly valuable in such situations. The feasibility of this concept was shown in [14].

As previously mentioned, an agent acquires capabilities based on the tasks assigned to it. Thus, the agent should be able to look for capabilities and acquire them from its peers. Also, it should be able to manage the capability life cycle (start, stop, restart or replace with another capability etc.). For this, capabilities have to be described in a manner that allows agents to search for and reason about them (i.e. match tasks to capabilities and compare between alternatives and evaluate the “fitness” of a capability to a task). For example, computationally heavy tasks could have capabilities optimized for different hardware platforms. Capabilities could similarly describe themselves as memory/CPU efficient for use in resource limited environments. It should also be possible to ascertain the trustworthiness of capabilities to guard against potential security threats. In previous research on compositionally adaptive agents, component (i.e. capability) description and matching has not been given prominence and emphasis was not given to defining components in a manner suited for reuse in a wider scope. However, there has been much research on component description in other areas of agent research (e.g. LARKS [25]) as well as in fields such as web services and service-oriented systems (e.g. WSDL, ontology) which VERSAG agents can make use of.

We see that capabilities need to be built in accordance to a standard model which allows agents to look for, use, store, discard and exchange them. Next, we identify several requirements VERSAG has of a capability model. The agent should be able to install, uninstall, start and stop capabilities without adversely affecting the functioning of the agent or other capabilities executing on the agent. Capabilities should also be able to communicate with each other and access services provided by the agent. Since application-specific capabilities would be developed by third parties, a well-defined contract according to which components can be developed is essential. There should also be minimum limitations set on the components being developed. The flexibility to have capabilities of different sizes and features, and being lightweight to suit resource constrained environments is also important. As capabilities would be updated with time, the ability to transparently upgrade a capability would be further enhanced if multiple versions of the same capability could coexist on an agent. This would also be useful to manage conflicting dependency requirements of capabilities (e.g. Capabilities X and Y respectively need versions 0.9 and 1.1 of capability Z). Finally, being reusable components, if capabilities could be built to adhere to widely used standards they could be used in a wider environment and components available in the wider environment could be used in VERSAG.

2.2.4 VERSAG Implementation

We have currently implemented the VERSAG framework on the JADE [26] agent platform. Previous works on compositionally adaptive agents have preferred to develop their own component models and mechanisms. While a custom-developed component model would be the simplest first approach, to meet some of the advanced requirements previously described would consume considerable time and effort. Therefore, in the current implementation of VERSAG we use a component model based on the OSGi framework [27].

Due to the flexible nature of the VERSAG model, we expect that it could be used in widely varying application scenarios. For example, an agent could start off as a personal agent and evolve itself to do different tasks such as online shopping, appointment scheduling, information retrieval and filtering and even be a universal remote controller for household devices simply by acquiring the appropriate capabilities required to function in each role. Next we describe such a scenario.

2.3 Application Scenario

John is a retired person who lives by himself. He is not technology-savvy, but has a personal assistant software agent that “lives” on his mobile phone and helps him with his daily activities. The agent maintains John's personal calendar and reminds him at appropriate times of appointments and events. For example, the agent just reminded John of his youngest granddaughter's birthday next week and the need of a gift. John requests the agent to buy a doll that would suit a six year old. The agent accepts the task and migrates to the home network server to do its gift search. Once on the home server, it makes a list of online stores by asking a yellow-pages web service and retrieves quotes from some of the stores’ via their web services. Finding an auction site for a popular brand of dolls, the agent decides to check it out too. It migrates to the auction site and by bidding manages to purchase a doll at a bargain price. Then, the agent migrates back to John's mobile phone via the home server, and informs John of the purchase. Later in the evening, the agent learns from the home context service that John is getting ready to go out. The agent reminds him to wear the wrist health monitoring device, which it notices to be not within communicating distance. John follows the advice of his trusted agent and steps out to enjoy a nice and safe walk.

This real-life application scenario describes a personal assistant performing diverse tasks on behalf of its human owner. We now briefly look at how a VERSAG based approach would support the agent in its diverse tasks. First, John’s request to purchase a gift would generate a new itinerary for the agent. While we do not elaborate on the itinerary building mechanisms, one possibility is that a generic “shopping itinerary” could be acquired and customized to the task at hand. Before it can start the gift search, the agent needs to acquire capabilities which would allow it to access web services, request and interpret price quotes, bid in an auction etc. The auction bidding capability may be acquired at the auction site from fellow bidder agents. The agent would also need to discard all these newly acquired capabilities before it migrates back to the mobile phone. We also see that it is able to carry out health monitoring activities when on the mobile phone by acquiring a capability which allows it to interpret data retrieved from a health monitoring device.
3. NETWORK COST MODEL

Before moving onto the network load analysis, we briefly explain different mobile agent migration strategies and how VERSAG capability-exchange works on top of it.

3.1 Agent Mobility

A mobile agent is generally viewed as consisting of code, data and an execution state [11]. When an agent migrates, these components have to be transferred to the destination agency. However, most agent toolkits (especially Java based ones) are not able to transfer the agent’s execution state. Hence they are considered as providing weak mobility while systems allowing transfer of the execution state are considered as providing strong mobility. In JADE, and therefore in our implementation, agent migration consists of transferring only the agent’s code and its data. The approaches used for this purpose by different agent toolkits are called migration strategies in [11]. Migration strategies can be broadly categorised as push or pull based on whether code (i.e. agent’s classes) is pushed to the destination by the origin agency or pulled by the destination agency. Further classification is possible considering the units of code being transferred (i.e. all at once, unit-by-unit) and the locations concerned. There are also approaches where only a specification of the agent is migrated (E.g. agent blueprints in the Dutch Agent Factory [21]). The destination agency is then able to rebuild the agent based on this specification making use of components the agency contains.

The JADE platform uses a pull-per-class strategy, where the destination agency pulls classes from the agent’s home agency as and when required [28]. Agent data is transferred as part of the object state when the agent migrates.

3.2 VERSAG Agent Migration

![Figure 4. Example of VERSAG agent migration process on the JADE platform](image)

From a technical point of view, all VERSAG agents are similar because they are made up of the same code base (which makes up the framework). Differences in behaviour are achieved by executing various capabilities. Capabilities are carried by an agent in its repository and constitute part of its data. Therefore, when a JADE based VERSAG agent migrates, most of its functionality is migrated as data. With peer-capability exchange, the agent is able to decide for itself on an optimum capability transfer approach. Figure 4 illustrates a scenario where a VERSAG agent migrates to a new location.

Agent A has 3 capabilities: cap1, cap2 and cap3, of which cap2 is already used and no longer needed. Cap3 is also available with agent B which is at a closer location to the destination agency. Therefore agent A first unloads cap2 and cap3 and then migrates to agency 2 carrying cap1 with it. The agent’s base classes are loaded from its home agency as per the JADE migration strategy. However, if a VERSAG agent had previously been to agency 2, these classes would be already available and would not have to be re-acquired. Finally A requests cap3 from B and acquires it. Steps 1, 5 and 6 shown in the diagram are carried out by VERSAG whereas the others form the standard JADE migration process.

3.3 Network Load Analysis

In this sub-section we focus on building an analytical model of the network load generated by a VERSAG agent migrating over a number of locations. We consider the total network traffic generated by an agent (to execute a given scenario) as the measurement of network cost.

The scenario considered is of a mobile agent sequentially migrating to a set of locations, executing different tasks at each location and finishing off at the originating point. The agent’s starting and terminating point is identified as node N0, which is also its home location. At each intermediate location Ni the agent has to perform a task taski which is only required at that location. The final task, taskn is performed at node N0 after having traversed all nodes. We identify the migration from node Ni-1 to Ni as migi. The final migration from node Nn-1 to N0 is known as mign.

![Figure 5. Agent has to migrate over a number of locations executing different tasks at each location](image)

The following symbols are used to represent components of the cost formula.
\[ B^C_X \] Represents the class size of a unit X

\[ B_{\text{Obj}}^X \] Represents the size of serialized unit X

\[ B^{\text{Overhd}}_{\text{mig}} \] The overhead due to agent migration, such as message exchanges between the agencies.

\[ B^{\text{Overhd}}_{\text{c-ex}} \] The overhead due to capability exchange includes the overhead for search, discovery and acquisition of capabilities.

We calculate the cost for two types of mobile agents used to execute this scenario: a normal mobile agent and a VERSAG agent.

Case I: A normal agent has all the functionalities required to execute the tasks coded into it.

Case II: A VERSAG agent is utilized to fulfill the itinerary. Task logics are implemented as capabilities. The capability handling policy is as follows. At each location the agent acquires the necessary capability, uses it and discards before migration. Hence, the agent always migrates as an empty (shell-only) agent. Another VERSAG agent residing at the home location (node N0) acts as the capability provider.

It is assumed that the overheads for migration and capability exchange are constant values. The size of the accumulated result that is carried along with the agent is considered to be negligible. We also assume that the total size of classes implementing the tasks in the normal mobile agent is equal to the sum of capability sizes in the VERSAG agent.

**CASE I**

The cost of a single agent migration is represented as the sum of agent’s class size, size of serialized agent and migration overheads. However, when the agent migrates to its home location, there is no need to send the agent classes since they are already available at the destination [11].

\[
B_{\text{mig},i} = \begin{cases} 
B_{\text{Obj}}^X + B^{\text{Overhd}}_{\text{mig}} & \text{if } i < n \\
B_{\text{Obj}}^X + B^{\text{Overhd}}_{\text{Mig}} & \text{if } i = n 
\end{cases}
\]  

We further breakdown the agent’s classes as those of the base agent and those of the various capabilities. That is,

\[
B^C_{\text{Ma}} = B^C_{\text{cap},i} \sum_{i=1}^{n} 
\]

Total network load for the normal agent can be represented as follows.

\[
B_{\text{Total}}^I = B_{\text{mig},1} + B_{\text{mig},2} + \cdots + B_{\text{mig},n} 
\]

\[
B_{\text{Total}}^I = (n-1) \left( B^C_{\text{ma}} \sum_{i=1}^{n} + B^{\text{Obj}}_{\text{ma}} \sum_{i=1}^{n} + B^{\text{Overhd}}_{\text{Mig}} \right) 
\]

\[
B_{\text{Total}}^I = (n-1) B^C_{\text{ma}} \sum_{i=1}^{n} + nB^{\text{Obj}}_{\text{ma}} \sum_{i=1}^{n} + nB^{\text{Overhd}}_{\text{Mig}} 
\]

**CASE II**

Total network load for the VERSAG agent includes the cost of \( n \) migrations and the cost of \((n-1)\) capability exchanges.

\[
B^I_{\text{Total}} = (B_{\text{mig},1} + B_{\text{mig},2} + \cdots + B_{\text{mig},n}) 
\]

\[
+ \left( B_{\text{c-ex},1} + B_{\text{c-ex},2} + \cdots + B_{\text{c-ex},n-1} \right) 
\]

The cost of a capability exchange is the class size of the capability and the exchange overhead

\[
B_{\text{c-ex},i} = (B^{\text{cap},i} + B^{\text{Overhd}}_{\text{c-ex}}) \quad \text{where } i = 1..(n-1) 
\]

Total load is then calculated as:

\[
B^I_{\text{Total}} = (n-1) \left( B^C_{\text{v}} + B^{\text{Obj}}_{\text{v}} + B^{\text{Overhd}}_{\text{v}} \right) + 
\]

\[
\left( B^{\text{Obj}}_{\text{v}} + B^{\text{Overhd}}_{\text{Mig}} \right) + \left( B^{\text{Obj}}_{\text{cap},i} + B^{\text{Overhd}}_{\text{c-ex}} \right) + \cdots 
\]

\[
B^{\text{Obj}}_{\text{cap},i} + B^{\text{Overhd}}_{\text{c-ex}} 
\]

\[
B^I_{\text{Total}} = (n-1) B^C_{\text{v}} + nB^{\text{Obj}}_{\text{v}} + nB^{\text{Overhd}}_{\text{Mig}} 
\]

\[
+ B^C_{\text{sum cap},i} \sum_{i=1}^{n} 
\]

For VERSAG to produce lower network load than a normal mobile agent, we should have \( B^I_{\text{Total}} \leq B^I_{\text{Total}} \). Using equations (3) and (5) we then arrive at the following.

\[
(n-1)B^C_{\text{v}} + nB^{\text{Obj}}_{\text{v}} + (n-1)B^{\text{Overhd}}_{\text{c-ex}} \leq (n-1)B^C_{\text{ma}} + nB^{\text{Obj}}_{\text{ma}} \sum_{i=1}^{n} + (n-2)B^{\text{Overhd}}_{\text{cap},i} 
\]

Thus we observe that the comparative agent code sizes, the size of capabilities, and capability exchange overhead are the key factors deciding which approach produces lesser network load.

### 4. EXPERIMENTAL EVALUATION

This section describes experiments carried out to verify the network load cost model developed previously.

#### 4.1 The Effects of JADE

We consider an implementation of the scenario described in figure 5 using the JADE based VERSAG prototype. The JADE approach to agent migration, discussed earlier, has several implications on the standard mobile agent migration model used to build the cost model.

- Agent code transfer happens from the home agency rather than from the agent’s previous location.
- Only classes that are used at the current location are transferred.
- Class-by-class migration increases the migration overhead in comparison to mechanisms where a collection of classes are migrated together as a single unit.

Since we are considering traffic generated over the whole network rather than between specific nodes, the change in the origin point of code transfers does not affect our model. However, two approaches were used to overcome the effect of the pull-per-class...
strategy. First, all custom developed classes were instantiated in the agent’s main class to ensure that they are used at each location. For third-party libraries required by the different tasks, they are artificially carried as agent data in the normal mobile agent (i.e. Case I).

4.2 Experiment Details
The experiments were carried out in a high-speed Local Area Network, using computers running Windows and Solaris operating systems and Java SDK 1.6.0. The agent toolkit used is JADE version 3.5 [29].

The agent tasks in the experiment consist of reading a collection of files at each location and extracting some information (i.e. word count) from them. The results are collected at each location and displayed at the final location. It is expected that at each location different types of files have to be read and therefore the agent requires a different capability at each location. The normal agent (Case I) contains the logic to read all types of files in it. For Case II, the ability to read a particular file type is represented as a single capability. The VERSAG agent migrates without any capabilities and only acquires a capability when it is needed. Immediately after use, the capability is discarded before migrating to the next location.

The number of locations, n, was 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 normal agent (Case I) and a VERSAG agent (Case II) were allowed to complete the itinerary. The agent platform was restarted between two test runs to ensure that any code caching mechanisms do not affect the readings. Total network traffic generated and time taken to complete the itinerary were measured.

4.3 Results and Discussion
The mean network loads and completion times observed are shown in table 1.

<table>
<thead>
<tr>
<th>n</th>
<th>Case I (normal)</th>
<th>Case II (VERSAG)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MB</td>
<td>Sec</td>
</tr>
<tr>
<td>2</td>
<td>3.53</td>
<td>1.21</td>
</tr>
<tr>
<td>3</td>
<td>13.87</td>
<td>4.33</td>
</tr>
<tr>
<td>4</td>
<td>31.20</td>
<td>7.05</td>
</tr>
<tr>
<td>5</td>
<td>55.23</td>
<td>12.88</td>
</tr>
<tr>
<td>6</td>
<td>86.35</td>
<td>17.14</td>
</tr>
<tr>
<td>7</td>
<td>124.10</td>
<td>24.12</td>
</tr>
<tr>
<td>8</td>
<td>169.03</td>
<td>33.37</td>
</tr>
<tr>
<td>9</td>
<td>219.83</td>
<td>42.36</td>
</tr>
<tr>
<td>10</td>
<td>279.18</td>
<td>50.50</td>
</tr>
</tbody>
</table>

Figure 6 graphically illustrates network load variation with increasing number of locations. It clearly indicates that VERSAG agents generate less network load in comparison to a normal mobile agent as the number of locations traversed increases. This can be understood by comparing equations 3 and 5. The base agent class and object sizes and the overheads are constants for this experiment while the number of locations n, class size of all capabilities \( B^C_{\sum_{i=1}^m} \), and object size of all capabilities \( B^O_{\sum_{i=1}^m} \) are variable. The equation for a normal agent (equation 3) contains products of the variable terms while the equation for VERSAG (equation 5) only contains the variable terms in isolation. Therefore, as observed above, the plot for a normal agent should display a quadratic increase whereas the plot for VERSAG only increases linearly.

Our test scenario assumes that a particular capability is required at exactly one location. Let us relax this condition and consider that there are instead \( m \) capabilities which could possibly be used over the \( n \) locations. The normal agent would still be compelled to have all the \( m \) capabilities embedded in it. In the case of the VERSAG agent however, if the agent continues to discard each capability after use, it could end up requesting the same capability multiple times. In this situation it could be beneficial for the agent to retain a used capability if it is likely to be needed again soon. Determination of the most beneficial approach for capability handling in such a situation would depend on multiple criteria such as, the probability that a capability would be required again, number of hops between successive uses of the same capability, cost of links to be traversed, ease of reacquiring the capability at another location and throughput requirements of the application. For example, it was shown in [14] that it is beneficial to discard capabilities when travelling over expensive links if they can be acquired with little effort at the destination. Thus, to reap the full benefits of the capability-based approach, agents would need to possess intelligent capability handling mechanisms.

Our network cost model only considered the network traffic generated by the agents. Another common and important cost criterion is round-trip time. While we did not formally analyse the
cost components, the readings in table 1 show that for the given experiment, the normal agent consumes more time as the number of locations increase. This behaviour is however particular to the current implementation. The time taken by VERSAG agents would increase as more complex protocols are used for capability discovery and acquisition and also depend on external factors such as network bandwidth, latency, resources available at different agent locations and agent platform features.

A fundamental advantage that our runtime adaptive agents have over a normal mobile agent is the ability to acquire new behaviours for situations which are not anticipated at design time. Thus, they would be able to continue execution in situations where a normal 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.

5. CONCLUSIONS

In this paper, we described our approach to runtime compositional adaptation of mobile agents for use in pervasive environments. The reference architecture of the agent framework and details of the reusable components, capabilities, were explained. We then briefly described mobile agent migration strategies and how our agent migration works on top of these existing strategies.

The main contribution of this paper is the analysis of network load generated by a VERSAG agent and identifying the various cost components. An experimental evaluation was then carried out to measure the actual performance of an adaptive mobile agent (VERSAG) against a conventional mobile agent. The experimental results allowed us to verify the accuracy of our cost model. We also identified further enhancements to increase the efficiency of VERSAG agents by using more intelligent capability handling policies.

In future we intend to build a model of the response time which would help us further improve our solution and also to experiment with intelligent capability handling mechanisms in a real-life application scenario using VERSAG.

6. REFERENCES


