Cybernetics and Systems: An International Journal

Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ucbs20

IMPROVING EFFICIENCY OF SERVICE-ORIENTED CONTEXT-DRIVEN SOFTWARE AGENTS

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

[a] Faculty of Information Technology, Monash University, Caulfield East, VIC, Australia

[b] Department of Computer Science & Computer Engineering, La Trobe University, Bundoora, VIC, Australia

[c] Faculty of Information Technology, Monash University, Caufield East, VIC, Australia, and ICT Centre, CSIRO, Canberra, Australia

Version of record first published: 11 Aug 2011.

To cite this article: Kutila Gunasekera , Seng Wai Loke , Arkady Zaslavsky & Shonali Krishnaswamy (2011): IMPROVING EFFICIENCY OF SERVICE-ORIENTED CONTEXT-DRIVEN SOFTWARE AGENTS, Cybernetics and Systems: An International Journal, 42:5, 324-340

To link to this article: http://dx.doi.org/10.1080/01969722.2011.595336

Cybernetics and Systems: An International Journal, 42:324–340

ISSN: 0196-9722 print/1087-6553 online
DOI: 10.1080/01969722.2011.595336

Improving Efficiency of Service-Oriented Context-Driven Software Agents

KUTILA GUNASEKERA[1], SENG WAI LOKE[2], ARKADY ZASLAVSKY[3], and SHONALI KRISHNASWAMY[1]

[1]*Faculty of Information Technology, Monash University, Caulfield East, VIC, Australia*
[2]*Department of Computer Science & Computer Engineering, La Trobe University, Bundoora, VIC, Australia*
[3]*Faculty of Information Technology, Monash University, Caufield East, VIC, Australia, and ICT Centre, CSIRO, Canberra, Australia*

The case for integrating software agent and web service paradigms has been well documented, and we believe that convergence of these two paradigms, enhanced with context awareness, can enable more efficient and effective pervasive services. Software agents in service-oriented environments have traditionally been limited to either using, providing, or aggregating services. We propose that in dynamic heterogeneous environments it would be sometimes beneficial if the agent, in addition to invoking remote services, could acquire the capacity to execute functionality provided by the service and run it locally. To this end, we build a performance analysis model that compares time consumption and network load of service access with that of component use. We argue that such a model would allow an agent to dynamically select the more efficient alternative. We present a multicriteria decision-making model that helps dynamic selection, describe experiments comparing the two approaches, and discuss results and lessons learned.

KEYWORDS context-awareness, service oriented computing, software agents

Address correspondence to Kutila Gunasekera, Faculty of Information Technology, Monash University, Str. 900, Dandenong Road, Caulfield East, Caulfield, VIC 3145, Australia. E-mail: kutila.gunasekera@infotech.monash.edu.au

INTRODUCTION

The case for integrating the software agent paradigm with the web services paradigm has been well documented in the literature (Richards et al. 2004; Bellifemine et al. 2007), with one of the main benefits being the use of software agents to dynamically aggregate services. We believe that the convergence of these two paradigms, enhanced with context awareness, can lead to more efficient and effective service-oriented systems. Today's mobile computing devices are also seen as consumers and providers of services (Berger et al. 2003; Medman 2006). Thus, software agents that produce and consume these mobile services need to be context aware to successfully execute in the face of changing environmental conditions.

The integration of web service and agent paradigms would allow agents to access web services and web services to access agent services. Also, in Richards et al. (2004), an agent factory for composing agents is used to compose web services. Our focus in this article is on situations where an agent is the end consumer or an intermediary that composes complex services using other web services. Traditionally, the two main approaches of allowing agents to access web services are either using an intermediary gateway agent to expose web service functionality as agent services to other agents (e.g., JADE Web Service Integration Gateway; Bellifemine et al. 2007) or for each agent to directly invoke web services (e.g., JADE Web Services Dynamic Client add-on; Scagliotti and Caire 2009). In the first approach, individual agents are unburdened with web service know-how, but the gateway agent could be a performance bottleneck. The second approach removes this bottleneck but adds the requirement that each agent should be able to discover and invoke web services. Selection of either approach would generally be based on the merits and demerits of each application situation.

With the current explosion in the number of mobile and embedded devices and advances in wireless communication technologies, mobile devices have become consumers as well as providers of services. Berger et al. (2003) identified that the major advantage of these services is the ability to reduce the need for user attention by program-to-program interaction. However, when a service is run on a battery-powered mobile device, it is faced with challenges such as poor processing capacity, irregular network connectivity, and an increased need to be energy aware (Bojic et al. 2010). Previous and ongoing research investigates methods to overcome these challenges. For example, the work by Preuveneers and Berbers (2008) attempted to increase availability and accessibility of services by smart service mobility, and Dustdar and Juszczyk (2007) proposed a solution based on dynamic replication and synchronization of web services in mobile ad hoc networks. Similar to the service provider, the users of such services need to be able

to take into consideration and deal with such eventualities. The solution we describe in this article is one such approach using context-aware service-oriented software agents.

Web service invocation could involve significant amounts of data being exchanged between the provider and client (e.g., in data mining and information retrieval domains). This can lead to huge communication costs and performance issues when accessing mobile services over relatively low-bandwidth and unreliable networks. Also, in the mobile service space, when the service provider itself is a mobile device, it is more likely that service provision would undergo frequent degradations or failures when the provider is faced with adverse conditions such as high load, poor connectivity, or low resources.

We propose that in such situations it would be beneficial for the client agent to acquire the web service's functionality and execute it locally instead of requesting the service to carry out the task. Such an alternative is beneficial when the cost of web service invocation is significantly higher compared to that of running the service locally. In particular, we believe that such situations could be encountered in the dynamic environments occupied by mobile services. For example, consider a mobile agent traversing fixed and mobile computing devices located within a building to discover context sources and collect data from them. Because it may encounter different types of data sources and formats, the agent depends on remote web services to process the data. Though this is generally a good approach to keep the agent lightweight and to avoid consuming too much device resources, when the volume of data is quite high, local processing after acquiring the necessary reasoning logic may be cheaper than transferring the data to a remote service. We also note that this approach could be beneficial for nonagent mobile web service clients. In this article, we build a model to analyze the performance of our proposed approach in comparison to the traditional approach of agent web service interaction. Our model could be used by software agents to dynamically select the more efficient approach to fulfill its goals. To this end, we also describe a multicriteria decision-making model that helps agents make this selection by taking into account contextual data. Experiments that compare the two approaches and performance of the multicriteria decision-making approach are presented and their implications discussed.

We previously proposed and implemented the VERsatile Self-adaptive AGents (VERSAG) agent framework (Gunasekera, Zaslavsky et al. 2009) as a novel approach to build dynamically adaptive mobile software agents. The network load behavior of VERSAG agents was analyzed and it was shown that they can achieve higher network efficiency compared to nonadaptive mobile agents. In this article we report the use of VERSAG agents in the service-oriented paradigm to achieve greater efficiency through dynamic processing scheme selections.

The rest of this article is structured as follows. The next section describes our model to analyze performance of the two approaches of service use and component use by agents. Then, we look at some previous research in compositionally adaptive mobile agents and provide a brief introduction to our own work in the area. Next, we present a case study scenario and briefly describe a model for dynamic selection of an efficient processing scheme based on contextual input. Finally, we describe our experiments, discuss the results, and conclude the article.

A PERFORMANCE ANALYSIS OF SERVICE USE VS. COMPONENT USE

In this section, we build an analytical model that investigates the performance of service use with that of component use for an identical task. The performance metrics used are *time* consumed and *network load* generated while fulfilling the given task. The scenario consists of a software agent on a computing device that is given a task to process a number of data items (located on the same device) that are assumed to be of nonnegligible size. The agent does not have the know-how to process the data by itself and has two options to follow. In Case I, the agent uses a remotely located (web) service to process data. For each datum, a service request encapsulating the datum is sent and a result obtained until all the data are processed (Figure 1(a)). In Case II, the agent acquires a component that has the know-how to process the data from a remote provider. The agent then repeatedly executes this component locally to process the data (Figure 1(b)).

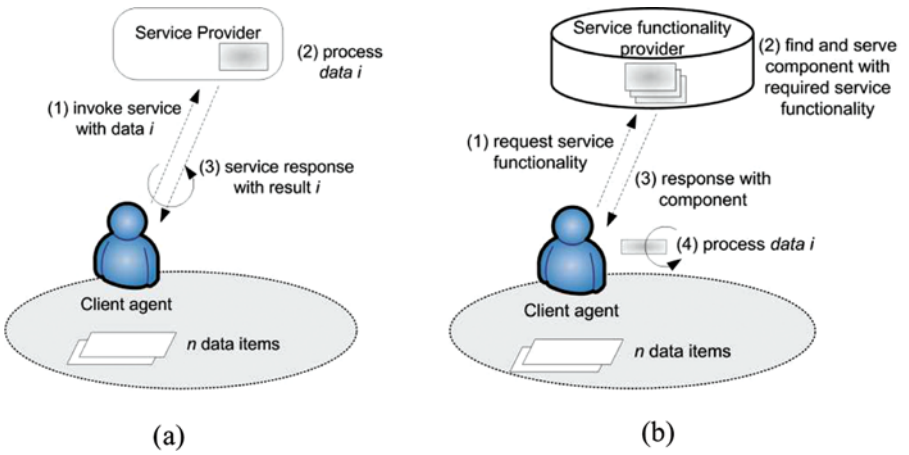

FIGURE 1 (a) Service use: the agent uses a remote service to process data; (b) component use: the agent acquires the service functionality as a component and processes the data at its own location (color figure available online).

The following symbols are used to build the analytical model. The number of data items is represented by n and the latency and bandwidth of the network link are represented by λ and bw, respectively. The size of data item i is B_i^d, B_i^{res} represents the size of the result from processing data item i. The overhead of a single request or response is represented as B^{ovrhd}. B^{cap} is the size of a component and B^{creq} is the size of a command requesting a component. Tp_i^l shows the time needed to process data item i at location l. T^{start_p} is the time needed to start processing at the agent's site.

We also make the following assumptions. The message overhead due to transmission control protocol (TCP) headers, fragmentation, retransmissions, etc., is fixed. The time to marshal and unmarshal data, delays due to network scheduling, and any waiting times before service processing starts are considered to be negligible. The time to start executing the component at the agent's site is fixed but nonnegligible and is therefore included in the model. The time T to move B bytes over the network link can be represented as follows:

$$T = \lambda + \frac{B}{bw}. \tag{1}$$

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 ($T_i^{serv\_res}$). The time taken to process all of the data is the sum of time taken to process each datum.

$$T_I = \sum_{i=1}^{n} (T_i^{serv\_req} + T_i^{serv\_res} + Tp_i^{svr}). \tag{2}$$

From Eq. (1) we have

$$T_i^{serv\_req} = \lambda + \frac{(B_i^d + B^{ovrhd})}{bw}$$

and

$$T_i^{serv\_res} = \lambda + \frac{(B_i^{res} + B^{ovrhd})}{bw}$$

Substituting these in Eq. (2) we get

$$T_I = 2n\left(\lambda + \frac{B^{ovrhd}}{bw}\right) + \frac{1}{bw}\sum_{i=1}^{n}(B_i^d + B_i^{res}) + \sum_{i=1}^{n} Tp_i^{svr} \tag{3}$$

Case II: Here, the total time taken 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 of the data items locally.

$$T_{II} = T^{cap\_req} + T^{cap\_res} + T^{start\_p} + \sum_{i=1}^{n} Tp_i^{loc}. \tag{4}$$

From Eq. (1) we have

$$T^{cap\_req} = \lambda + \frac{(B^{creq} + B^{ovrhd})}{bw}$$

and

$$T^{cap\_res} = \lambda + \frac{(B^{cap} + B^{ovrhd})}{bw}$$

Substituting these in Eq. (4) we get

$$T_{II} = 2\left(\lambda + \frac{B^{ovrhd}}{bw}\right) + \frac{(B^{creq} + B^{cap})}{bw} + T^{start\_p} + \sum_{i=1}^{n} Tp_i^{loc} \tag{5}$$

From Eqs. (3) and (5) it can be seen that time efficiency of the two approaches depends on network parameters (bandwidth and latency), data (number of items and total size), and processing (processing times and size of processing component). In a high-bandwidth and low-latency network, total processing time ($\sum_{i=1}^{n} Tp_i^{svr}$ or $\sum_{i=1}^{n} Tp_i^{loc}$) would be the significant component for both service use and component use cases. In contrast, in a low-bandwidth, high-latency network, the other components of data size and component size would come into consideration. For component use (Case II), there is no direct influence from increasing data size, except that it would most likely lead to longer processing times.

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 and the response contains the corresponding result. Together with the messaging overheads, this is expressed in Eq. (6). For Case II, generated network traffic consists of the request from the client and the response encapsulating the component, as shown in Eq. (7).

$$B_I = 2nB^{ovrhd} + \sum_{i=1}^{n} (B_i^d + B_i^{res}). \tag{6}$$

$$B_{II} = 2B^{ovrhd} + B^{cap} + B^{creq} \tag{7}$$

It can be seen that though network load depends on the number and size of data items in Case I, in Case II it is independent of data and is mainly made up of the component being transferred.

COMPOSITIONALLY ADAPTIVE AGENTS

To realize the solution put forward in this article, we need an agent implementation that allows agents to compositionally adapt their structure at runtime. In this section, we briefly describe some previous research that investigate compositionally adaptive mobile software agents and then introduce our approach to compositionally adaptive mobile software agents.

Related Work

Though there has been a considerable amount of research on developing adaptive agent systems recently (Marín and Mehandjiev 2006), in most of these works, individual agents are not adaptive, and adaptivity is achieved at the system level through mechanisms such as agent interactions, migration, and learning. The work on dynamic agents by Chen et al. (1999) views individual agents as carriers of components that provide the agent with intelligence. When faced with tasks that require functionalities beyond what it possesses, a dynamic agent is able to download new programs from a remote uniform resource locator (URL). However, the agent has to be explicitly informed what components to obtain and does not have the ability to dynamically evaluate its performance and adapt if required. Agents in the generic adaptive mobile agent (GAMA) architecture (Amara-Hachmi and Fallah-Seghrouchni 2005) are composed of multiple components and are capable of adapting themselves to suit new environments. However, a GAMA agent can select components only from what is available at its current location and only adapts immediately after migration. The open source Java Agent DEvelopment Framework (JADE) platform (Bellifemine et al. 2007) also provides rudimentary support for runtime adaptation of agent functionality through a JADE-specific application programming interface (API) that provides limited control of the adaptation process. The port-based adaptable agent architecture (PB3A) by Dixon et al. (2000) is built with a port-based module (PBM) as the basic building block and has a port-based agent as a self-adaptive structure. The component architecture for service agents (CASA) by Sesseler and Albayrak (2001) also contain agents that exhibit compositional adaptivity to some degree. Though these systems provide various levels of support for compositionally adaptive mobile software agents, we have proposed an approach that attempts to further advance the state-of-the-art with a flexible component-based solution to

enable building intelligent adaptive agents suited for diverse tasks and environments.

VERSAG Agent Framework

VERSAG (Gunasekera, Zaslavsky et al. 2009) is a novel approach to engineer adaptive mobile software agents that are context-driven and can execute on dynamic heterogeneous environments. A salient feature of VERSAG is the ability of agents to acquire new functionality from peer agents without depending on designated providers. An agent's useful functionality is provided in the form of reusable software components termed *capabilities*. A capability is agent platform independent and can be attached to and detached from a software agent to provide the agent with a particular behavior. Further, an agent is context driven and can execute on dynamic heterogeneous environments. Its component-based structure allows it to have different functionalities and agent architectures embedded in it during its lifetime, increasing the agent's versatility. A VERSAG agent's primary task is to execute an itinerary that specifies a list of locations the agent has to visit and activities to carry out at each location. To carry out an activity, the agent may need multiple capabilities. Because it is self-adaptive, the agent decides when and from where the necessary capabilities are acquired and discarded. Intelligence for decision making is also provided to the agent in the form of capabilities. Therefore, it is possible for a VERSAG agent to be intelligent and self-adaptive when required and to be a simple and lightweight itinerant agent when advanced intelligence and adaptation are not required.

VERSAG agents are built on top of existing agent toolkits and thereby are deployable on existing agent-based infrastructure. Figure 2 shows the structure of a prototype VERSAG agent based on the JADE agent toolkit. Here, the JADE agent is a small layer that interfaces between the agent platform services and the upper functionalities provided by VERSAG components. The *kernel* is the agent's core and provides it with the itinerant behavior. The *itinerary execution service* provides the kernel with necessary methods to interpret itinerary commands. Application-specific capabilities of

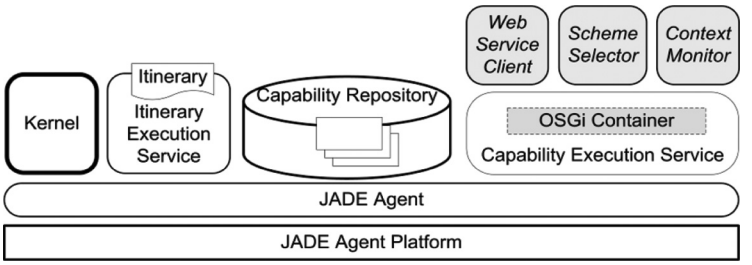

FIGURE 2 Structure of a JADE-based VERSAG agent.

the agent are stored in the *capability repository*. The *capability execution service* provides means to load, run, and stop capabilities that are available in the repository. In the prototype, capabilities are defined according to the OSGi framework (OSGi Alliance 2003), allowing them to be reused wherever a compatible OSGi container is available. Further details of the VERSAG agent internal architecture, algorithms, and implementation details can be found in Gunasekera, Krishnaswamy et al. (2009). We find that VERSAG agents are particularly suited to implement the described service-oriented software agents due to their ability to change structure at runtime, ability to acquire components from peers, and context-driven nature, which allows them to dynamically evaluate the environment and adapt accordingly.

IMPLEMENTATION OF DYNAMIC SCHEME SELECTION USING VERSAG

Equations (1) to (7) show that performance of the two schemes considered depends on multiple criteria and that preselection of one scheme over the other may not be the most efficient. Thus, agents need to be able to dynamically select the best scheme based on contextual information. An agent should also be able to find and acquire new functionality at runtime such that it can retrieve the web service's functionality and execute it locally. Here we provide a brief description of using VERSAG to implement such a scenario.

Case Study Scenario

We assume that a VERSAG agent playing the role of an information gathering agent is assigned an itinerary where it is required to migrate to a number of locations (i.e., computing devices) and process various types of data found on them. The agent does not *a priori* have the necessary data processing capabilities and can only decide on the processing needed after a preliminary inspection of data at each location. 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.

Once the required processing is identified, the agent has to select an appropriate capability with which to fulfill the task. It has two capabilities to select from, either a capability that implements the necessary processing logic or a web service client capability that lets it pass the data to a remote web service for processing. Though the choice is between two capabilities from the agent's perspective, from the application's perspective it is a choice

between component use and service use. The agent is equipped with a *scheme selector* capability that helps it to select between these two schemes. The inputs needed for scheme selection are network parameters, web service service-level agreements (SLAs) and details of data (i.e., type, size, speed of generation). Network information is obtained from a separate context-monitoring capability and 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. If the local processing scheme is selected, the agent acquires the data processing capability from a provider agent that is colocated with the remote web service. The VERSAG agent shown in Figure 2 is equipped with context monitoring, scheme selection, and web service client capabilities.

In a task such as sensor data extraction, selecting a scheme before commencing execution may not be feasible due to lack of sufficient knowledge up front. In such cases, the scheme selector could select an initial scheme without rigorous preselection, monitor task progress, and instruct the agent to switch schemes if needed. There is also the possibility that dynamic changes in the environment could lead to a selected scheme becoming inefficient mid-way through execution. Continuous generation of data by a sensor, changes in the network, or a change in the status of the web service are several such changes. In these situations too, the agent could switch its selected scheme through the scheme selector.

Scheme Selector Capability

The aim of the scheme selector is to aid the agent in selecting the lowest cost alternative when it has several groups of capabilities with which to fulfill a given activity. It is assumed that to fulfill an activity, the agent has to execute one or more capabilities in sequence. Thus, each alternative consists of several capabilities. The scheme selector implements a multicriteria decision-making model to sort the available capability groups in increasing order of cost. Input to the cost model consists of cost criteria, the relative importance of each criterion, any limits on cost values, and a list of available capability groups that can be used to fulfill the activity. Time required for an activity, generated network load, memory/CPU requirements, capability accuracy, security, probability of reuse, monetary costs, and user preferences are some of the possible cost criteria. The current implementation limits cost criteria to time consumption and network load. The relative importance of cost criteria can be explicitly specified by the user. First, the cost of each capability group is evaluated against any limits and groups that fail to meet the necessary constraints discarded. Next, the weights of cost criteria are normalized to build a *priority vector*. For remaining capability groups, costs are estimated for elements that were assigned weights. The reciprocals of estimated cost values are used to create a *utility matrix*, where utility is an indicator of the benefit

that can be gained by selecting a particular group. Thus, the lower the estimated cost, the higher the utility gained. Rows in the utility matrix represent capability groups and columns represent cost criteria. The utility matrix is multiplied with the priority vector to arrive at a vector containing aggregate utility values. The capability group with highest utility is then selected by the agent.

When applying the scheme selector to our current scenario, we use time as the only constraint and do not place any upper limit on time consumption. Also, our capability group size is one. The time estimate for a group with only one capability is the sum of the time to acquire that capability and time to execute it. Equations (4) and (5) (Case II) further elaborate this as applicable to the current scenario. Time to run the web service client capability is elaborated in Eqs. (2) and (3) (Case I). The capability acquisition and data transfer (to and from web service) times can be estimated once network parameters and capability and data sizes are known. We assume that the result from the web service is the same size as the data sent. The scheme selector obtains network parameters from the context monitoring capability, which measures actual network bandwidth and latency available to the agent. It is assumed that time to process data is proportional to data size. Thus, capability execution time can be expressed as follows:

$$Tp_i^{loc} = (size\ of\ data\ item\ i) \times (processing\ speed). \tag{8}$$

For the web service client capability, processing speed is part of the web service's SLA and is made available as part of the capability metadata. For the capability that implements the processing logic, speed also varies depending on the computing capacity of the location at which it is executing. Thus, capability metadata could provide a formula for calculating processing speed based on computing capacity. For experimental evaluations described in the next section, processing speed was calculated in advance and included in capability metadata.

EXPERIMENTAL EVALUATION

Experiments were conducted in order to evaluate our performance analysis model by comparing the performance of service use and component use approaches under varying conditions. For these 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 description earlier, it is assumed that the agent is unaware of the type of files prior to arriving at the device. The processing involved is to convert the files to PDF format. The agent already has with it a capability that allows it to access a SOAP-based *txt2pdf* web service that can do the conversion. This web

service accepts a single text file in a Simple Object Access Protocol (SOAP) request, converts it to a PDF, and sends it back to the client encapsulated in a response message. The agent has to sequentially request the web service to convert files. A component use alternative in the form of a *txt2pdf* capability is also available with a peer agent at the same location as the *txt2pdf* web service. The agent's itinerary requires it to minimize time taken for its tasks. Thus, the agent uses its scheme selector capability to search for and select the best alternative from the above two.

Our test environment is as follows. The web service is deployed on a desktop PC with Windows XP and 2 GB memory, and the client agent runs on a slower notebook PC with 3 GB memory and Windows Vista. Both computers have Java SDK 1.6.0 and the agent toolkit used is JADE v. 3.7 with the LEAP add-on (Bellifemine et al. 2007). The desktop PC is connected to a local area network (LAN) that has high-speed Internet connectivity. The notebook PC is connected first over an IEEE 802.11 g wireless local area network (WLAN) in a university laboratory and subsequently over a high-speed downlink packet access (HSDPA) network of an Australian telecom operator. The measured network latency in the WLAN environment was less than 0.5 ms and the observed bandwidth 15 Mbits/sec. Over the HSDPA network, the average latency was 158 ms and observed average bandwidth was 300 Kbits/s. The size of the *txt2pdf* capability is 1,070 KB.

We test the time taken to complete the processing as the size of data increases. We increase data size by increasing total file size from 250 to 3,000 KB while keeping the number of files fixed at 10. The time measured for web service use (Case I) approach is from the start of execution of the client capability to its completion. In the component use (Case II) approach, because the agent has to acquire the capability, we also measure the 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.

When the experiment was conducted on the WLAN network, the scheme selector consistently selected the web service (Case I) approach, whereas on the HSDPA network it selected the component use (Case II) approach for the majority of the data sizes. Table 1 shows the utility values computed for 3,000 KB data on the two networks. The approach with higher value is selected by the agent.

TABLE 1 Scheme Selector Computed Utility Values (Average Values Shown) in the Two Networks for 3,000 KB Data Files

Network	Utility of web service use (I)	Utility of component use (II)
WLAN	0.58	0.42
HSDPA	0.29	0.71

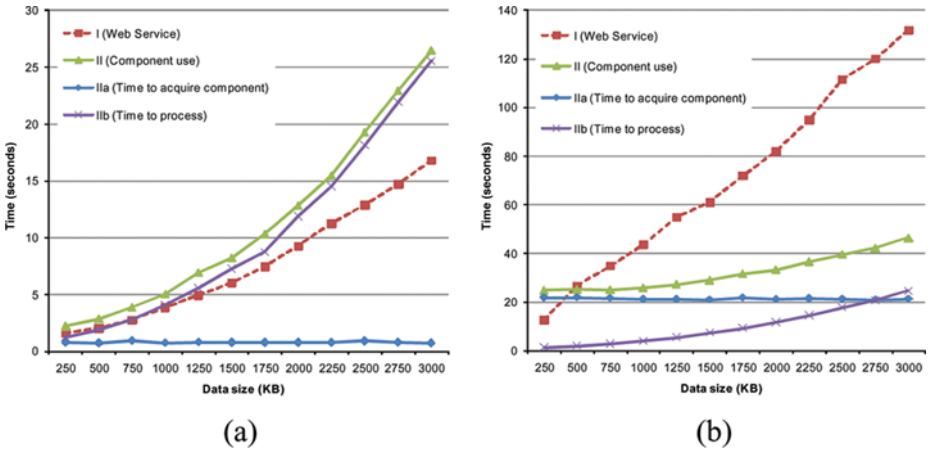

FIGURE 3 Response time variation with increasing data size: (a) in WLAN network and (b) in HSDPA network (for both graphs, the number of files is 10 and average values were used to plot graphs) (color figure available online).

The experiments were repeated by altering the agent's itinerary to bypass the scheme selector and preselect an approach. In Figure 3, we plot the observed variation in response time as total data size increases for both approaches. Figure 3(a) is on the WLAN network and shows that the web service (I) approach consistently consumes less time than the component use (II) approach. Figure 3(b) for the HSDPA network shows that the component use (II) approach consumes less time for all cases except 250 KB data size. The behavior of these plots confirms that the scheme selector correctly predicted and selected the lowest cost approach.

In Figure 3(a) (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 behavior is consistent with the discussions based on Eqs. (3) and (5) where it was identified that in a network with high bandwidth (bw) and low latency (λ) the total processing time $\left(\sum_{i=1}^{n} Tp_i^{loc}\right.$ or $\left.\sum_{i=1}^{n} Tp_i^{svr}\right)$ would be the significant component. 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 s on desktop PC to 30.7 s on a notebook for 3,000 KB of data) such that its performance was far worse than the component use (II) approach.

Figure 3(b) shows response time variation when the tests were run on the HSDPA network. 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, and increases in the component use approach occur only as a result of increases in processing time.

The Wilcoxon signed-rank test was used 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 4 we show network load variation corresponding to the two test scenarios. It should be noted here that network load was not used as a selection criterion. 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 behavior is common across both networks (WLAN and HSDPA) and is consistent with observations made based on Eqs. (6) and (7).

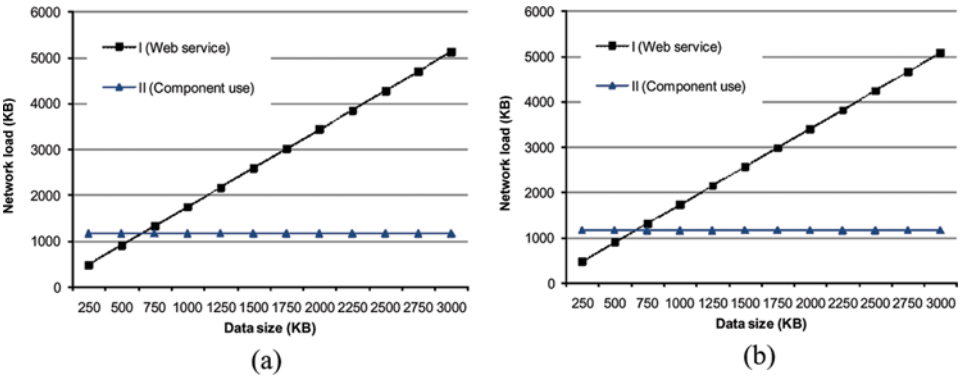

FIGURE 4 Network traffic generation with increasing data size in (a) a WLAN network and (b) an HSDPA network (average) (color figure available online).

Based on these experimental results we observe that the web service approach consumes less time compared to the component use approach when executing on a high-bandwidth network. In contrast, on a low-bandwidth network, as the size of data to be processed increases, the component use approach becomes more attractive because 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, whereas 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 we did not consider in our experiments, is the computational capacity of the two devices (where the agent and web service execute). We conclude, based on our experimental results, that in some situations it can be beneficial for a software agent to acquire a software component and execute it locally instead of asking a remote provider to execute the service on its behalf.

Given information about its environment, including bandwidth, computational capacity of each location, size of data to process, as well as size of the capability (if it is not already available with the agent), the agent can dynamically choose the better scheme to improve its efficiency in terms of relevant criteria. This also means that if the agent had chosen the web service scheme to start with, but during processing the network performance degrades (e.g., the user walks out from a WiFi area to a wide-area lower bandwidth connection), the agent could switch schemes in order to maintain performance levels.

CONCLUSIONS AND FUTURE WORK

In service-oriented computing, software agents have traditionally been viewed as either service providers or service composers. In this article, we argued that software agents can achieve greater efficiency and usability by being able to dynamically select between web service use and component use (i.e., acquiring the service's functionality and executing it locally). We presented a performance analysis model of the two approaches using time consumption and network load as performance indicators. It was identified from the performance analysis model that network parameters, data, and processing are the three main factors affecting the performance of each model. We described experimental evaluations that illustrated that the two approaches can be beneficial under different circumstances. Thus, we have solidified our argument that software agents enhanced with context awareness can perform better by selecting the more efficient approach from between service use and component use based on the situation.

We also described a multicriteria decision-making model, the scheme selector, that helps compositionally adaptive context-aware software agents to select from among multiple alternatives to increase its efficiency in terms of desired parameters such as execution time, generated network load, result accuracy, and resource consumption. A brief discussion of this model and an application scenario motivating its benefits were provided. Experimental evaluations using our prototype implementation of the scheme selector confirmed that it is capable of accurately selecting between the two alternatives (web service use and component use) by taking into account web service parameters, size of data, and network parameters.

Security, privacy, and related issues are of significant importance in mobile agent research. Though we acknowledge the significance of these issues in relation to VERSAG agents and our proposed solution, they are outside the scope of the current research and therefore have not been discussed. We are currently engaged in further evaluations to test the scalability of our approach in environments with large numbers of agents. Implementing and evaluating support for dynamic scheme selection after processing commences is also being investigated. As future work, we plan to implement a case study scenario to demonstrate the benefits of VERSAG agents playing multiple roles in real-life heterogeneous distributed environments.

REFERENCES

Amara-Hachmi, N. and Fallah-Seghrouchni, A. E. 2005. Towards a generic architecture for self-adaptive mobile agents. Paper read at European Workshop on Adaptive Agents and Multi-Agent Systems, Paris, France, March 21–22, 2005.

Bellifemine, F. L., Caire, G., and Greenwood, D. 2007. *Developing multi-agent systems with JADE*. Chichester, England: John Wiley & Sons.

Berger, S., McFaddin, S., Narayanaswami, C., and Raghunath, M. 2003. Web services on mobile devices—Implementation and experiences. In *Proceedings of the 5th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2003)*, IEEE Computer Society, Monterey, CA.

Bojic, I., Podobnik, V., and Kusek, M. 2010. Agent-enabled collaborative downloading: Towards energy-efficient provisioning of group-oriented services. *Lecture Notes in Computer Science* 6071: 62–71.

Chen, Q., Chundi, P., Dayal, U., and Hsu, M. 1999. Dynamic agents. *International Journal of Cooperative Information Systems* 8(2–3): 195–223.

Dixon, K. R., Pham, T. Q., and Khosla, P. K. 2000. Port-based adaptable agent architecture. *Lecture Notes in Computer Science* 1936: 181–198.

Dustdar, S. and Juszczyk, L. 2007. Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks. *Service Oriented Computing and Applications* 1(1): 19–33.

Gunasekera, K., Krishnaswamy, S., Loke, S. W., and Zaslavsky, A. 2009. Runtime efficiency of adaptive mobile software agents in pervasive computing environ-

ments. In *Proceedings of the ACM International Conference on Pervasive Services (ICPS'09)*, London: ACM Press.

Gunasekera, K., Zaslavsky, A., Krishnaswamy, S., and Loke, S. W. 2009. Component based approach for composing adaptive mobile agents. *Lecture Notes in Computer Science* 5559: 90–99.

Marín, C. A. and Mehandjiev, N. 2006. A classification framework of adaptation in multi-agent systems. *Lecture Notes in Computer Science* 4149: 198–212.

Medman, N. 2006. Doing your own thing on the Net. *Ericsson Business Review* 2006: 48–53.

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

Preuveneers, D. and Berbers, Y. 2008. Pervasive services on the move: Smart service diffusion on the OSGi framework. *Lecture Notes in Computer Science* 5061: 46–60.

Richards, D., van Splunter, S., Brazier, F. M. T., and Sabou, M. 2004. Composing web services using an agent factory. In *Extending web services technologies—the use of multi-agent approaches*, Editors: Lawrence Caredon, Zakaria Maamar, David Martin Boualem Benatallab. New York: Springer.

Scagliotti, E. and Caire, G. 2009. *Web services dynamic client guide*. Accessed June 30, 2009. http://jade.tilab.com/doc/tutorials/DynamicClientGuide.pdf

Sesseler, R. and Albayrak, S. 2001. Service-ware framework for developing 3G mobile services, in proceedings of the 16th International Symposium on Computer and Information Sciences (ICSIS XVI), Edited by S. Kuru, F. Ince, and H. L. Akin, 5–7 November, at Antalya, Turkey, Isik University Publications.