Adaptation Support for Agent Based Pervasive Systems

Kutila Gunasekera[1], Shonali Krishnaswamy[1],
Seng Wai Loke[2], and Arkady Zaslavsky[1,3]

[1] Faculty of Information Technology, Monash University, Australia
{kutila.gunasekera,shonali.krishnaswamy}@monash.edu
[2] Department of Computer Science & Computer Engineering, La Trobe University, Australia
S.Loke@latrobe.edu.au
[3] Luleå University of Technology, Sweden
arkady.zaslavsky@ltu.se

Abstract. Pervasive computing systems execute in dynamic highly variable environments and need software that are context-aware and can adapt at runtime. Mobile agents are viewed as an enabling technology for building software for such environments due to their flexibility, migratory nature and scalability. This paper presents a novel approach which aims to further enhance this advantage by building compositionally adaptive mobile software agents that are also context-driven, component-based and have the ability to exchange their components with peer agents. We present the formal underpinnings of our approach and a decision making model which assists agent adaptation. We also describe our current implementation and experimental results to evaluate the benefits of the proposed approach.

1 Introduction

As we move towards the era of *invisible computers* envisioned by Weiser [1], computers are becoming smaller and increasingly pervasive in day-to-day environments. Pervasive computers now have to execute in diverse and rapidly changing environments where low powered devices and wireless communication media are *de rigueur*. The miniaturization and portability of devices have also contributed to these devices having lower computing capacity in comparison to their stationary counterparts.

Developments in pervasive computing have been primarily driven by advances in hardware and communication technologies, whereas application development for such environments has lagged behind considerably. Pervasive applications need to be adaptive and versatile to survive rapidly changing environments and requirements. How to build such applications is a current challenge in pervasive computing research [2]. Self-adaptive software, autonomic computing and mobile software agents [3] are amongst some of the approaches seen as attractive options for pervasive application development. Flexibility, scalability and ability to reduce complexity by delegation are some of the desirable features that mobile agents bring to pervasive computing applications [4]. Pervasive applications are also likely to require mobility - a salient feature of mobile agents – in order to support mobile users as well as to migrate when faced with intermittent connectivity and device resource constraints. The ability of a

P. Sénac, M. Ott, and A. Seneviratne (Eds.): MobiQuitous 2010, LNICST 73, pp. 90–101, 2012.

statically defined agent to survive in an uncertain environment is limited. Thus, adaptive agent systems are being investigated to overcome this limitation [5].

Our research explores the use of compositionally adaptive software agents to build applications for pervasive environments. In our proposed approach, agents are lightweight, mobile and can autonomously adapt based on contextual input [6]. We aim to provide adaptations in agent capabilities in a manner that accommodates dynamic changes, with flexibility that is unprecedented compared to earlier work. This we see as important in pervasive environments where actual resources available at a particular location cannot be determined *a priori*, so that dynamic uptake and exchange of capabilities become useful. Context-awareness is another major requirement in pervasive applications [2] as they should be able to sense changes in the environment and adapt accordingly. For example, an application may need to monitor the power level of its current device and migrate to another node if the level falls below a threshold. Adaptation decisions are affected by multiple criteria such as resource availability, application semantics and user preferences. This paper discusses a multi-attribute cost model to assist agents in making dynamic adaptation decisions based on contextual and user given criteria. We formally define our cost model and present preliminary evaluation results which illustrate the benefits of the model. In particular, we show that an agent, using our cost model, and having access to contextual information about parameters of the model, can make accurate estimates of performance with different agent capabilities, and so, can make effective adaptation decisions at run-time.

The rest of the paper is structured as follows. In section 2 we present a conceptual overview of our compositionally adaptive agents. Section 3 describes the formal underpinnings of the approach and the adaptation cost model. A brief introduction to our prototype implementation is given next, followed by experimental evaluations in section 5 and some related work in section 6. Finally, we conclude in section 7.

2 Conceptual Overview

We propose a novel approach to develop smart pervasive applications through the use of dynamic compositionally adaptive mobile software agents [6]. The proposed VERSAG (VERsatile Self-adaptive AGents) agents are context-driven, adapt by acquiring new software components at runtime, and execute on dynamic heterogeneous environments. An agent's component-based structure allows it to have different architectures embedded within during its lifetime, and its useful functionality is provided in the form of reusable software components termed *Capabilities*. Two salient features 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. An agent's high-level task is to execute an *itinerary* assigned to it where an itinerary specifies a list of *locations* it has to traverse and activities to execute at each location. To carry out an activity, the agent may need multiple capabilities. It decides when and from where the necessary capabilities are acquired and 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.

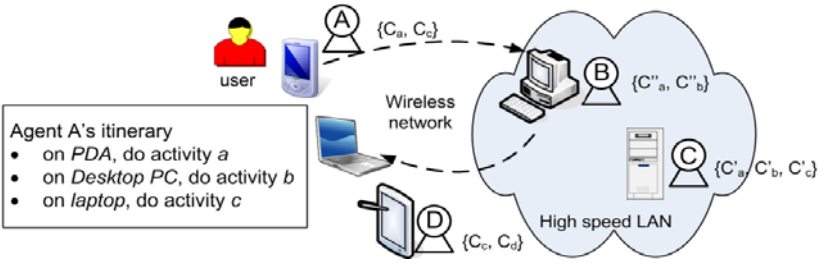

Fig. 1. A hypothetical scenario showing VERSAG agents deployed in a pervasive environment. The capabilities contained in each agent are shown beside it within braces, and the subscript corresponds to an activity that the capability can carry out.

Figure 1 illustrates a hypothetical scenario of how this approach can be useful in a pervasive computing environment. A human user requests his personal agent *A,* which resides on his PDA, to do a task. This task is converted into an itinerary as shown. The agent first does activity *a* using capability C_a. Then, discards its capabilities and migrates without carrying anything. On the desktop PC, it asks nearby agents for capabilities to do activity *b*. Agents *B* and *C* respond with C''_b and C'_b. *A* selects C''_b which is nearer, executes it, and then discards it before moving to the laptop. Once on the laptop, *A* searches for C_c to do activity *c*. *D* and *C* respond and *D*'s response is selected because it is more suitable for resource-constrained devices.

If the agent has a choice of capabilities to select from, it needs to be able to select those which minimize execution *cost*. Time required to carry out the activity, generated network load and reliability, memory/CPU requirements, component accuracy, security, probability of reuse, monetary costs and user preferences are some of the many possible criteria which make up the cost. While many of these criteria are interrelated, it is neither necessary nor realistic to consider each one before a decision making step. Thus, this research limits itself to supporting the following main criteria: time, network load, maximum memory needs, CPU usage and level of accuracy of execution. Which costs should be minimized for a given situation can be explicitly specified by the user or inferred from environmental conditions.

Specific adaptation actions of an agent include stopping and starting a capability, discarding a capability, acquiring a new capability, migrating to a different location or terminating itself. Two other forms of adaptation an agent could undertake are changing its base implementation to support migration to a different platform or changing the order of visits in its itinerary. These two forms of adaptation are outside the scope of this research and are therefore deferred for further investigation.

3 The Formal Model

Having provided an overview of our proposal, in this section we formally define the key concepts that underpin our approach. We also describe how an agent executes an activity at a particular location and the adaptation cost model.

Definition 1: Adaptive agent
Let I be an agent's itinerary, $O = \{move, terminate, get\_c, drop\_c, start\_c, stop\_c\}$ be the set of operations an agent can perform, with $*\_c$ operations performed on capabilities and, C be the set of capabilities that the agent is carrying. Then an agent A can be defined as a tuple of the form (I, O, C).

Definition 2: Capability
A capability is a central concept in our approach and provides an agent with application specific behaviours. Let U be a set of unique identifiers, F be the set of functionalities that the capability contains, $Y = \{primitive, compound\}$ be a set of capability types, E be the set of environments on which the capability can execute and M represent meta-data about the capability such as owner, version, security certificates, algorithms/units used and optimizations. Then, a capability is a tuple of the form (u, F, y, E, M) where $u \in U$ and $y \in Y$. Functionality of a capability could range from a simple database query execution to providing the agent with a new architecture such as the BDI model.

Definition 3: Capability Specification
A capability specification (spec) is used to identify capabilities that can fulfil a given set of functions and can execute in a given set of environments. Let F represent the set of functions the capability contains and E define the set of environments on which the capability can execute. A capability spec is then a tuple of the form (F, E).

Given a capability spec $cs = (F_s, Env_s)$ and a capability $c = (u, F, y, Env, M)$, capability c will *match* the spec cs if and only if $F_s \subseteq F$ and $Env_s \subseteq Env$. That is, for a capability to *match* a capability spec, the spec's full set of functionalities and supported environments must be supported by the capability.

Definition 4: Activity
An activity is the unit of work an agent has to carry out at a particular location. Multiple capabilities may need to be executed in sequence to carry out the activity. Let CS be a set of capability specs and O be a set of operations $\{get\_c, drop\_c, start\_c, stop\_c\}$. A tuple (cs, o) where $cs \in CS$ and $o \in O$ represents application of an operation on a capability. An activity A is defined as a finite sequence $(cs_1, o_1), (cs_2, o_2), \ldots, (cs_n, o_n)$ where $cs_i \in CS$, $o_i \in O$, $i \in \{1, 2, .., n\}$ and n is the number of capabilities in the activity.

A *location* is a place an agent can visit and contains the necessary runtime environment for the agent to exist. Various computational resources are available at a location. A capability can only use resources available at its current location. *Cost elements* are used to represent different types of costs incurred during itinerary execution and have integer values. *Constraints* place limits on cost elements and can be associated with a location, an itinerary, an itinerary step, a capability or an agent. *Relative constraints* are generally user specified and used to indicate relative weights of cost elements. *Absolute constraints* place limits on permissible value ranges for cost elements.

Definition 5: Relative constraint
A relative constraint r is a finite sequence of the form $(q_1, w_1), \ldots, (q_i, w_i), \ldots, (q_n, w_n)$ where $q_i \in Q$ the set of cost elements and $w \in \mathbb{Z}$ represents the weight of that cost element.

Definition 6: Absolute constraint
An absolute constraint places an upper and lower limit on a cost element. It is a tuple of the form *(q, min, max)* where $q \in Q$ the set of cost elements, $min < max$ and $min, max \in \mathbb{Z}$ represent the lower and upper limits of the cost element. For example, *(time, 0, 10)* means the "time" cost element must have a value between 0 and 10.

Definition 7: Itinerary
An itinerary consists of a sequence of *itinerary steps* the agent has to carry out. An itinerary step is a tuple *(l, A, R)* where *l* is a location, *A* an activity and *R* a set of constraints. Let *L* be the total set of locations and *A* the set of activities. Then, an itinerary is defined as a finite sequence $(l_1, a_1, R_1), \ldots, (l_i, a_i, R_i), \ldots, (l_n, a_n, R_n)$ where $l_i \in L$, $a_i \in A$ and $i \in \{1, 2, .., n\}$. Activity $a_i \neq \varnothing$ while constraints can be empty.

While an itinerary is defined as above, it is expected that users would be issuing high-level commands to the system, which would be converted to detailed itineraries for agents to execute.

We next describe an agent's activity execution process. The following discussion assumes that the agent first migrates to the relevant location and then searches for and acquires capabilities required at that location. Therefore the activity fails if suitable capabilities could not be found. Figure 2 provides an algorithmic description of this process.

```
1 Let CS = {cs1, cs2…, csm} be the set of capability specs which can
  be used to carry out activity a
2 Search for capability instances matching CS
3 Combine capability instances to form groups of capabilities
  G = {g1, g2 …, gp} where each gi can fulfil activity a
  If G = ∅
     Fail
  End if
4 Build set of constraints CONS = {r1, r2 …, rn} from explicit and
  implicit constraints identified through context sensing.
5 Sort G in increasing order of cost By applying the adaptation cost
  model.
6 For each gi in G
     Acquire capability instances of gi by applying operation get_c()
     If acquisition successful
        Break loop
     End if
  End for
  If a completed group is not available
     Fail
  End if
7 Execute capabilities in correct sequence to fulfil activity
```

Fig. 2. Activity execution algorithm of an agent

Step 1: Identify sequences of capability specifications that can accomplish the given activity. It is possible that more than one such sequence exists. Activity to capability spec group mappings may be contained in the itinerary, stored with the agent or available from an external source.

Step 2: For each capability spec, search for matching capability instances in the agent's local repository and from peer agents. The search mechanism itself is implementation specific and is expected to be changeable.

Step 3: Combine received capability descriptions to build groups that can accomplish the given activity. If no groups can be formed, the activity has to fail. Group formation needs to take into account limitations of capability instances and their compatibility with each other.

Step 4: The agent senses environmental conditions at the current location and identifies applicable implicit constraints. These are combined with the explicitly specified constraints to build the complete set of constraints.

Step 5: Apply the *adaptation cost model* to sort the capability groups according to cost. Constraints, capability group details and available capability descriptions are used as input to the selection process. Groups that don't meet absolute constraints are removed.

Step 6: Select the least cost capability group and acquire the relevant capability instances (if they are not already with the agent). If acquisition fails due to any reason, acquire the next best capability group.

Step 7: Execute the acquired capabilities in correct sequence.

The adaptation cost model of step 5 above is illustrated in Figure 3.

```
1    Let Q = {q1, q2, … qk} be the set of cost elements
     relative constraint {(q1, w1), … (qm, wm)} where m ≤ k
     Cons = {con1,con2,… conn} be absolute constraints where n ≤ k
     G = {g1, g2, … gp} be the set of alternative groups
2    For each coni in Cons with cost element qi
         For each gj in G
             Estimate cost in terms of qi for gj
             If estimated cost does not satisfy coni
                 Remove gj from G
             End if
         End for
     End for
     If G = ∅
         Fail activity
     End if
3    From relative constraint, build normalized priority vector P_{k×1}
4    For each gi in G
         For each qj part of a relative constraint
             If cost estimate of gi not available
                 Estimate cost in terms of qj for gi
             End if
         End for
     End for
5    Build utility matrix U_{p×k} from group cost estimates
6    Calculate utility vector V_{p×1} ← U_{p×k} × P_{k×1}V
     Sort V in decreasing order of utility
```

Fig. 3. Adaptation cost model

The input to the cost model consists of cost criteria, relative constraint (provides weights of each criterion), absolute constraints and a list of available groups. Step 2 evaluates each capability group against the *absolute constraints* and discards groups that fail to meet the necessary constraints. If no groups remain at the end of this step, the process fails. Step 3 normalizes the weights from the relative constraint to build a normalized priority vector of the cost elements. In step 4, for remaining groups, costs are estimated for elements that are part of the relative constraints. Step 5 builds a utility matrix from these estimated costs. Given that there are *p* alternative groups remaining and *k* different cost criteria, the utility matrix is an *p x k* matrix where element u_{ij} is the reciprocal of the estimated cost value of group g_i for cost criterion q_j (except for "accuracy" criterion which is used as it is). The reciprocal of the cost value is used as an indicator of the benefit or "utility" that can be gained by selecting a particular group. Thus, lower the estimated cost, higher the utility gained. In step 6, by multiplying this utility matrix with the priority vector, which indicates the relative importance of each cost element, we obtain a vector which provides aggregate utilities for each group. The group with highest utility is the one that has lowest cost and is to be selected by the agent.

4 Implementation Details

This section presents the prototype implementation of the VERSAG concepts and theory previously discussed. It is built on top of the JADE agent toolkit [7] with an OSGi [8] based capability model. Figure 4 illustrates the structure of an agent. The agent, as previously mentioned is itinerary-driven at its core with the kernel driving this behavior. The capability repository stores the agent's 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. The capability exchange service (not in figure) fulfils the dual roles of a capability requestor and provider. In its provider role it listens for capability requests from peer agents and respond as appropriate. The requestor role gives an agent the ability to request capabilities from peers. The base JADE agent is the framework's point of contact with the underlying agent platform and provides access to services such as mobility and communication. Capabilities themselves are agent-platform agnostic and can

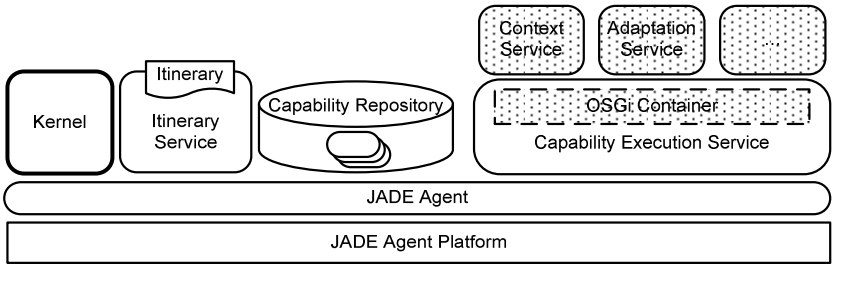

Fig. 4. Structure of a JADE based VERSAG agent

execute wherever a suitable execution environment is made available. Since the context service and adaptation service are implemented as capabilities, it is possible for a VERSAG agent to switch to different implementations of these services or be stripped of them to become a lightweight itinerant agent. An agent's reference architecture and more design details are described in [6].

This implementation limits itself to considering time, network load, maximum memory needs, CPU usage and level of accuracy as the cost criteria to be considered before adaptation. Itinerary execution *time* is often a crucial factor that needs to be minimized. It is also desirable to reduce *network load*, especially in wireless networks. Hence, these two are the primary cost elements supported. For a mobile agent, it is difficult to make design-time assumptions about the computational resources that would be available during its lifetime, as it traverses devices with different enabling opportunities. Thus, it is desirable if the agent can dynamically adapt itself to work with available computational resources. A VERSAG agent is able to achieve this by dynamically changing its constituent components. For example, an agent maybe required to execute an activity on a resource-constrained device, which has limited memory, processor and battery capacity. Available resources may be further limited by having to share them with a number of other agents. Memory and CPU cycles are two key resources that have to be thus shared. However, due to differences in computer architectures, hardware and system software, it is difficult to provide estimates of CPU or memory requirements of a software component in a useful platform-independent manner. Therefore, we use a *CPU usage ranking* which can be used as a relative indicator of a capability's processing needs. For memory, the maximum required *heap memory size* in the Java Virtual Machine is used as a criterion. In certain applications, it is desirable to generate an approximate result using fewer resources rather than a more accurate output which is considerably more resource-intensive. The cost element of *accuracy* is incorporated into the decision making model to reflect this requirement. Thus, an agent may decide to select a less accurate capability over a more accurate one in order to reduce overall costs.

5 Experimental Evaluation

This section describes experimental evaluations carried out to verify the performance of the agent adaptation cost model. The experiments were conducted with two main goals: to check whether an agent correctly selects the least cost alternative from among many, and to check whether the generated estimates are representative of the actual costs. We describe two sets of experiments: one using *network load* and the other using *time* consumed as the main cost criterion. No absolute constraints are specified in both cases. Three separate workloads representing a simple agent GUI, an information retrieval task and a file sorting task were used. Time based tests were conducted over a high-speed LAN and an IEEE 802.11g wireless network while a 3G wireless broadband link was also used for load based tests. The three test computers used each had comparable computational capacity and were running Windows XP, Vista and Ubuntu Linux. All had Java SDK 1.6.0 and JADE version 3.7 with the LEAP add-on was used. We first describe the formulae used by the prototype to estimate costs followed by the experimental results.

The total network load generated when executing an itinerary step consists of the load due to execution of the capabilities as well as the load generated when the agent searches for and acquires the relevant capabilities. Given that an alternative group has *n* capabilities, the total network cost of selecting that group is made up of the load generated during adaptation decision making (B^{search}), load generated during the actual acquisition of the capabilities (B_i^{acq}) and any network load generated during capability execution (B_i^{exec}). The total cost can be expressed as shown in equation 1.

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

Further, $B_i{}^{acq}$ is made up of request size (B^{creq}), capability size (B_i^{cap}) and the overhead of each request (B^{ovrhd}) as shown in equation 2. It is assumed that the size of a capability request and overhead of each request is constant for a given network.

$$B_i^{acq} = B^{creq} + B_i^{cap} + 2B^{ovrhd} . \quad (2)$$

The network cost of capability execution (B^{exec}) is functionality and implementation specific and it is expected that a fixed value or a formula to estimate the same will be provided with the capability's meta-data. For the experimental workloads used, network traffic generated during execution was nil.

For an alternative that has *n* capabilities; the time to sequentially acquire the capabilities and execute the itinerary step can be expressed as follows:

$$T = T^{search} + \sum_{i=1}^{n} \left(T_i^{cap\_req} + T_i^{cap\_res} + T_i^{start\_p} + Tp_i^{loc} \right) . \quad (3)$$

Here T^{search} is the time taken for the decision making process, $T_i^{cap\_req}$ the time to request a capability, $T_i^{cap\_res}$ the time to receive a capability in response, $T_i^{start\_p}$ the time to start execution of the capability, and Tp_i^{loc} the time to execute it at location *loc*. The time T, to move *B* bytes over a network link with latency λ and bandwidth *bw* can be represented as $T = \lambda + B/bw$. Using this in equation 3 we get:

$$T = T^{search} + \sum_{i=1}^{n} \left(2\lambda_i + \frac{(B_i{}^{cap} + B^{creq} + 2B^{ovrhd})}{bw_i} + T_i^{start\_p} + Tp_i^{loc} \right) . \quad (4)$$

Every prototype agent is equipped with a context-sensing capability which measures network parameters. While it could measure bandwidth and latency, for our experiments request size, overhead, load and time for decision making were estimated manually using trial runs and supplied to this context-sensing capability. For the wireless network, the context-sensing capability measured the average latency as 1ms and average bandwidth available to the 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 concluded that B^{search} can be estimated as $(3900 \times no.of\ provider\ agents)$. 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

makes these estimations. Of the needed capability meta-data, capability size was measured by the peer agent. Time to start a capability and execution time were also estimated in trial runs and added to its meta-data. During the experiments, Wireshark was used to measure the actual network load generated and code was instrumented to log actual time consumption.

Based on experimental data, the error between estimated and actual values was calculated as a percentage of the estimated value. For the load based experiments, mean percentage error was 7.17% with a maximum of 13.5%. For time based experiments, higher error percentages could be observed when the estimates were less than 2 seconds whereas it was less than 10% when the estimate was larger than 2 seconds. In figures 5 and 6 we compare the estimated and actual costs for the two sets of experiments. They clearly illustrate that our estimates are representative of the actual cost, 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.

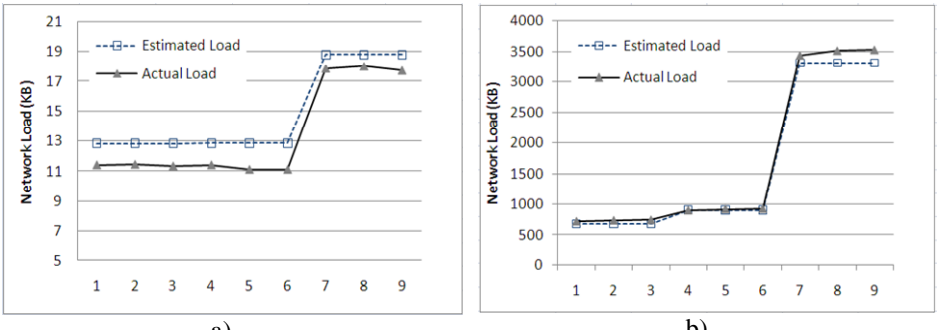

Fig. 5. Comparison of Actual and Estimated Loads for the tests, sorted in ascending order of estimate size. Two graphs are plotted for clarity, each graph containing 9 tests.

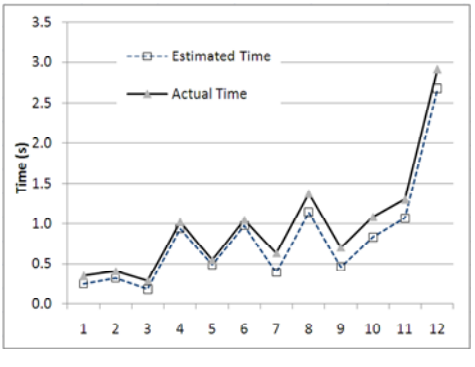

Fig. 6. Comparison of average Actual and Estimated time consumption. The first 6 tests were on a LAN environment and the following 6 conducted over a WLAN.

Table 1. Predicted utility values and actual costs for the three workloads, each with two available alternatives *a* and *b* are shown. Load and time values shown are averages.

		Network Load (KB)		Time (ms)	
Workload	Alternative	Utility	Actual	Utility	Actual
1	a	0.021	731.0	0.78	709
1	***b***	0.971	17.8	0.22	1089
2	a	0.214	3495.9	0.75	1306
2	***b***	0.786	913.5	0.25	2919
3	a	0.4995	11.2	0.90	638
3	***b***	0.5004	11.4	0.10	1378

In Table 1, we compare predicted utility values for the three workloads with actual measured values. Columns 3 and 4 are for the first set of experiments with *network load* as the constraint. For the first two workloads the agent correctly predicts higher utility values while in workload 3, the utilities are identical to three decimal places and the actual difference between the two workloads less than 200 bytes. In columns 5 and 6, *time* is the constraint and we see that all predictions are accurate. The results clearly indicate that the cost model is able to correctly select the least cost alternative for the agent to fulfil its goals.

While our experimental results validate the accuracy of the model, prediction accuracy is dependent on multiple factors. Primary among these are the meta-data provided by capability developer which includes time to start and execute a capability and network traffic generated by the executing capability. These are in turn likely to depend on other factors such as the runtime environment and data being processed. We expect that a capability's temporal and traffic generation behaviours will be tested on a standard environment and used to build the meta-data which in term could either be constant values or formulae for use by the adaptation cost model. These then need to be adapted by the agent to suit its current environment. Non-deterministic nature of the environment, especially the network, is another primary factor which can affect the prediction accuracy.

A major feature of our approach is the ability to simultaneously aggregate multiple criteria for decision making. The accuracy of cost aggregation has been separately tested using synthetic data and is not reported here. Our twin aims were to check whether the predictions are representative of the actual costs and whether the alternative with lowest cost is correctly selected. As per the experimental results, it is evident that the proposed approach achieves these two aims and can be beneficial for adaptation decision making in pervasive environments.

6 Related Work

We briefly list some previous research where mobile agents were applied to pervasive computing scenarios. GAMA (Generic Adaptive Mobile Agent architecture) [9] builds compositionally adaptive mobile agents from ground up for use in ubiquitous computing environments. The open source JADE agent platform [7] also provides rudimentary support for runtime adaptation of agent functionality. Preuveneers and Berbers [10] describe a non-agent solution to mitigate service disconnections in mobile ad hoc networks with dynamically migrating services. Further related research

is listed in [6]. Our approach aims to overcome various limitations found in these and to provide a flexible component based solution to build intelligent adaptive agents suited for pervasive environments.

7 Conclusions

Building software for pervasive computing environments is a current challenge in pervasive computing research. Our goal was to build an approach to address this challenge while making use of desirable features provided by the agent paradigm. To this end, we proposed a novel context-driven component based mobile agent framework. As contributions of this paper, we presented the formal underpinnings of the approach and described a multi-criteria decision making model which assists the agents make context and user preference driven adaptation decisions at runtime. A prototype implementation of this novel approach was also described.

Experimental evaluations were carried out using three different workloads on wired and wireless networks. The results confirm that our estimation strategies for time and network load are accurate and allow agents to effectively select the least cost alternative to carry out its given task when multiple options are present. We are currently engaged in further evaluations and intend to verify the scalability of our approach in environments with large numbers of agents and alternatives. As future work, we also plan to implement a case-study scenario to demonstrate the benefits of our approach in real-life pervasive computing applications.

References

1. Weiser, M.: The Computer for the 21st Century. Scientific American (1991)
2. Niemelä, E., Latvakoski, J.: Survey of requirements and solutions for ubiquitous software. In: 3rd International Conference on Mobile and Ubiquitous Multimedia, pp. 71–78. ACM, College Park (2004)
3. Cardoso, R.S., Kon, F.: Mobile Agents: A Key for Effective Pervasive Computing. In: ACM OOPSLA 2002 Workshop on Pervasive Computing, Seattle (2002)
4. Zaslavsky, A.: Mobile agents: can they assist with context awareness? In: IEEE International Conference on Mobile Data Management, pp. 304–305. IEEE Press, Los Alamitos (2004)
5. Marín, C.A., Mehandjiev, N.: A Classification Framework of Adaptation in Multi-Agent Systems. In: Klusch, M., Rovatsos, M., Payne, T.R. (eds.) CIA 2006. LNCS (LNAI), vol. 4149, pp. 198–212. Springer, Heidelberg (2006)
6. Gunasekera, K., Krishnaswamy, S., Loke, S.W., Zaslavsky, A.: Runtime Efficiency of Adaptive Mobile Software Agents in Pervasive Computing Environments. In: ACM International Conference on Pervasive Services (ICPS 2009), pp. 123–132. ACM, London (2009)
7. Jade - Java Agent DEvelopment Framework, `http://jade.tilab.com/`
8. About the OSGi Service Platform: Technical Whitepaper. OSGi Alliance, pp. 1–19 (2007)
9. Amara-Hachmi, N., Fallah-Seghrouchni, A.E.: Towards a Generic Architecture for Self-Adaptive Mobile Agents. In: European Workshop on Adaptive Agents and Multi-Agent Systems, Paris (2005)
10. Preuveneers, D., Berbers, Y.: Pervasive Services on the Move: Smart Service Diffusion on the OSGi Framework. In: Sandnes, F.E., Zhang, Y., Rong, C., Yang, L.T., Ma, J. (eds.) UIC 2008. LNCS, vol. 5061, pp. 46–60. Springer, Heidelberg (2008)