State-On-Demand Execution for Adaptive Component-based Mobile Agent Systems

Yuk Chow, Wenzhang Zhu, Cho-Li Wang, and Francis C.M. Lau
Department of Computer Science and Information Systems
The University of Hong Kong
Pokfulam, Hong Kong
{ychow,wzzhu,clwang,fcmlau}@csis.hku.hk

Abstract

The introduction of mobile code in the pervasive computing environment provides a good opportunity for research in ways to enhance execution flexibility. We note that current mobile code is too heavyweight and not adaptive enough to be used in pervasive computing where devices are resource-limited and heterogeneity is the norm. In this paper, we propose a new lightweight, component-based mobile agent system that can adapt to diverse devices and features resource saving as one of its aims. The system supports strong mobility of mobile code, which is a prerequisite for achieving system flexibility and good performance. The system discretize the transmission of code and execution states and relies on a scheme called state-on-demand (SOD) for the execution of the mobile code. We provide performance results to demonstrate the effectiveness of the SOD scheme.

1. Introduction

Information appliances (e.g., PDA and mobile phones) and many other ubiquitous devices will soon become an indispensable part of our daily living. Many different types of devices are taking advantage of wireless networks and the Internet to provide services to users. We have come to an age of *pervasive computing* [6], when people will be able to carry out computation at anytime, anywhere on any device.

Mobile agent (MA) is a special type of mobile code characterized by its peer-to-peer and autonomous nature [2]. It has been proved effective for supporting asynchronous computation in the distributed computing environment, especially in the presence of volatile mobile networks. It is therefore natural to exploit MA technologies in pervasive computing environments for adaptive and flexible computation on diverse devices.

Nevertheless, the inherent nature of pervasive computing environment has posed some serious challenges on the design of *mobile agent systems* (MASs). Some of these challenges stemmed from the resource constraints of these small and highly heterogeneous devices; much attention should be paid on conservation of resources such as network bandwidth and memory.

Many commercial MASs, such as Aglets [7], Voyagers [12], Grasshopper [1], etc., have been developed during the last decade. However, most of them paid little attention to issues such as lightweight execution and adaptiveness, the key elements to reap the benefits of MA technologies being applied to the pervasive computing world. They usually put their focus on application issues, such as usability of the agent platforms and compliance to MA standards.

As ubiquitous devices are getting more popular, some recent MASs are designed with a goal to run on these devices. For example, the University of West Florida had a plan to create a small mobile agent platform based on their previous work on NOMADS MAS and Aroma VM [8]. JADE-LEAP [3], a software framework for developing FIPA-compliant mobile agents using the LEAP library, provides a MAS that has a small memory footprint and is suitable for mobile lightweight applications developed based on J2ME-CLDC (Java 2 Platform Micro Edition-Connected Limited Device Configuration). However, adaptiveness to heterogeneous and the changing environment remains to be a major issue in these systems if they are to be deployed in the pervasive computing world.

In a pervasive computing environment, migration is an essential system service which enables applications to follow people wherever they move. Some projects have tried to address this service. One of them is the *one.world* project of the University of Washington [9]. It however provided only weak mobility support. Executing tasks were not allowed to migrate. This crippled the system's ability to support pervasive computing, e.g., to delegate execution when there are not enough resources on the device.

In this paper, we propose a new lightweight component-based MAS that can adapt to diverse devices. Our system aims at resource saving while supporting strong mobility of mobile code. The system discretize the transmission of code and execution state and relies on a scheme called *state-on-demand* (SOD) to execute the mobile code. We address three important issues in our system: adaptivity, resource saving, and strong mobility.

The rest of the paper is organized as follows. Section 2 describes the design of our MAS. Section 3 presents the implementation. Section 4 provides an in-depth evaluation of the system's performance. Section 5 studies the related work. Finally, we conclude in Section 6.

2. Design

In this section, we discuss our approaches to solving the problems of adaptivity, strong mobility, and the resource saving respectively in a MAS.

2.1. Adaptivity to devices

In a pervasive environment, mobile and stationary devices will dynamically connect and collaborate to help users to accomplish their tasks. In a pervasive computing environment, information flows more quickly and in more directions than traditional distributed computing. It is therefore imperative for software systems and applications to be able to adapt to changes in their execution environment on the fly, in order to provide a suitable and relatively stable working environment for the users.

For this vision to become a reality, developers must build applications that constantly adapt to a highly dynamic computing environment. Context-aware computing refers to the explicit ability of a software system to detect and respond to changes in its environment. Various adaptation techniques have been previously explored, from lower-level techniques of dynamically changing routing information to changing the fidelity (i.e. quality) of data. However, dynamically changing the way an application carries out its functions—*functionality adaptation*, is not well explored.

We propose to build a new MAS in Java with functionality adaptation, to support context-aware pervasive computing. Instead of a monolithic chunk, an application is assembled from disparate code components, called *facets*, which are brought in from the network and can be dynamically linked to the execution in accordance with the current context (e.g., locality, resource constraints, user's preference). An ontology-based knowledge representation and reasoning technique is used to specify the semantic features of facets upon their creation, and to automate facet selection and application construction. With functionality adaptation, a pervasive computing system is expected to be most flexible and adaptive, and can significantly enhance the mobility of pervasive applications.

2.2. Strong mobility

Strong mobility support is the natural need of mobile agents, which allows them to preserve the complete execution status after migration. Strong mobility enables execution delegation by migrating a task to computational servers when the client device does not have enough resources to carry out the execution.

In our system, the strong mobility support is realized by using a source-code preprocessor, which inserts additional code for state capturing and resuming. State capturing is to checkpoint the current execution state of a mobile agent. The execution state includes a series of Java frames, each containing the Java class name, the method signature and the activation record of the method. The activation record consists of the bytecode program counter (PC) and the local variables. The resuming process is to restore the execution state according to the captured state. This includes the restoration of both the variables, the PC, as well as the calling sequence in the execution state.

By "instrumenting" the application source code, the execution state can be captured into a list of frame objects through an exception-throwing mechanism, and resumed when the list is sent to its destination. Our system is based on the JavaGo software developed by the Tokyo University [10]. On top of JavaGo, we added some additional data structures, assumptions and code instrumentation schemes to support thread state migration. In the instrumentation scheme, we assume that all the running threads already have a reference to a *commonly shared object*. This shared object is to manipulate all the states related to inter-thread communications. Also, the activities of the threads are centrally co-ordinated by this object. For example, if a thread wants to trigger the migration event, it only needs to set a flag on the shared object. Upon detecting this change of flag value, all the threads would know that the migration is going to take place.

2.3. Resource saving

Resource saving refers to the reduction of memory and network bandwidth usage. We introduce and employ a technique called state-on-demand (SOD) execution.

During the execution of a MA in our MAS, only the *facet specifications* describing the required functionalities, rather than the actual facets, are stored in the container. The container provides an application-like feeling to the end-user who does not know how to program with container and facet abstractions. In that case, the user can invoke the container to bring up an interface. Based on the user's input, the con-

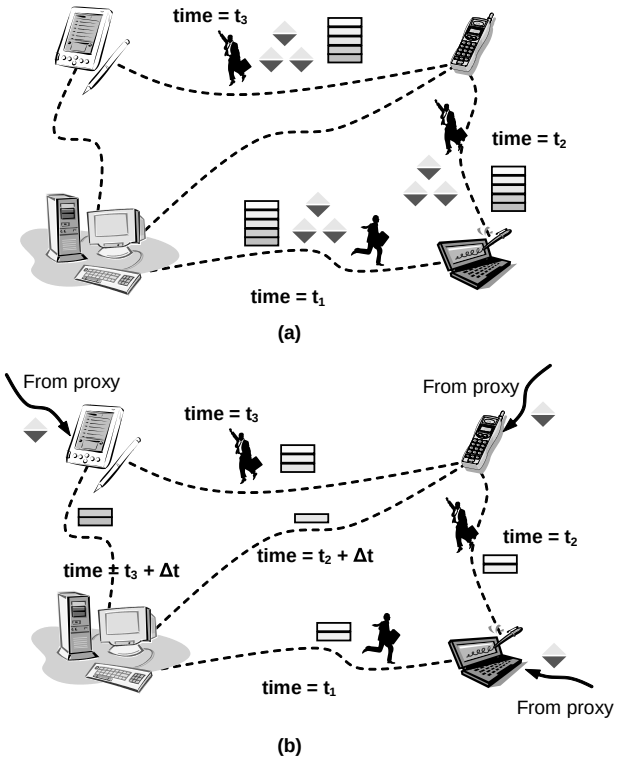

Figure 1. Migration model for (a) traditional MA and MAS; (b) our lightweight MAS.

tainer will make an appropriate facet request, which is sent to a proxy server for fetching the required facets. In this way, the memory and network bandwidth usage is saved because unnecessary facets will not be brought in.

Moreover, the structure and the First-In-Last-Out nature of the execution stack allow for further reduction in resource consumption. If we have a way to get rid of the "temporarily useless" execution states at the bottom of the stack and migrate/restore them only when they are required during execution, further reduction in bandwidth consumption is possible. This forms the basis of our SOD scheme.

Figure 1 illustrates the difference between a traditional MAS and our lightweight MAS. For most traditional MASs, all code and execution states are carried by the agent throughout their whole itinerary. As such, much bandwidth is wasted in migrating the unused bottom stack frames and the non-adaptive code. By contrast, our SOD scheme migrates only a portion of required stack frames for later resumption of the execution. This saves bandwidth consumption due to unnecessary transfers, especially when the mobile agent needs to traverse multiple hops. Furthermore, since only the top-

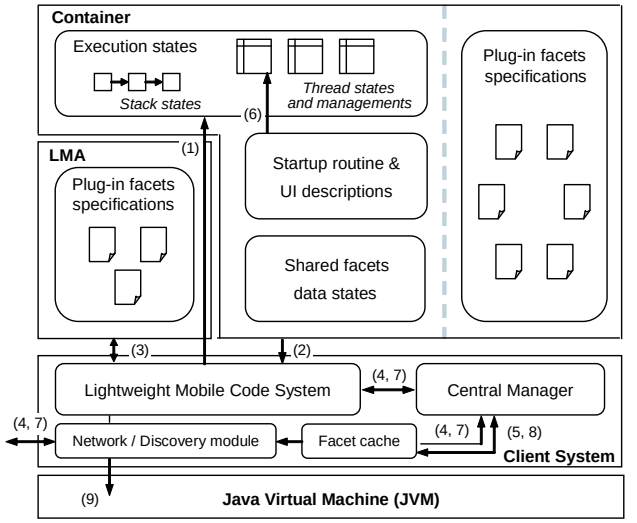

Figure 2. The system architecture.

most segment is needed in order to carry out the execution, and the required codes are brought in on demand, the memory usage in the target device is much reduced.

3. Implementation

We have implemented the Lightweight Mobile Agent System (LMAS) in Java (Sun Java SDK 1.3.1), and a source code instrumentation engine in Standard ML (SML/NJ) for supporting strong mobility and lightweight execution, which is derived by modifying the JavaGo source code preprocessor.

3.1. System architecture

Figure 2 shows the architecture of the LMAS. The system consists of a middleware system and two major kinds of entities. The *Lightweight Mobile Code System* (LMCS) is a middleware sitting on top of a native OS or virtual machine to handle the activities of the mobile agents it hosts, including creation, migration, and destruction of the agents. During execution, LMCS communicates with the central manager of the system in order to exercise the policies that govern the agents. *Strong mobility* and *state-on-demand* are supported in the LMCS. The two kinds of entities, *Lightweight Mobile Agent* (LMA) and the *container*, can be hosted in the LMCS. These two entities cooperate to function like a traditional mobile agent roaming from LMCS to LMCS.

Figure 2 also demonstrates the interactions among the various entities in our system. When a migration signal is raised, the execution state is captured by the LMCS and

stored inside the container (1). The LMCS then assigns an LMA to the container with its destination properly set (2, 3). At this time, the facets needed for carrying out basic agent operations are also fetched on demand (4, 5). The agents are then sent to the destination site. At the receiving end, the container is extracted and set to a "restoring state". With new threads created and re-registered (6), execution can be resumed by fetching suitable facets for their corresponding execution states (7, 8, 9). Since the capturing and resuming of states are totally done on demand at application level, our LMCS essentially supports strong mobility without affecting the portability of the system.

3.2. Facet

In the proposed software system, a functionality is embodied in a facet. A functionality can be considered a single well-defined task in an application. Essentially, functionality can be seen as a contract defining what should be done. The contract includes (1) input specification; (2) output specification; (3) description of what is carried out, i.e. what are valid outputs for a set of inputs; (4) pre-conditions: the ranges of input data size supported; (5) post-conditions: which values are nullified, error conditions; and (6) side effects: the requirement of I/O. The contract defines the functionality to be achieved, but not how it should be achieved. Implementations can use different algorithms, each with different performance characteristics or resource requirements, as long as they stick to the contract. Indeed, functionality defines the interface for interaction and is independent of the implementation.

The facet consists of two parts:

- **Shadow.** This is the place where the contract of the facet are defined. It includes information about the facet, for example, the facet id (facetID), the funcID of the functionality it achieves, input and output specification, vendor and versioning information, its resource requirements (such as the size of the working memory) and its functionality dependencies, and the charging scheme for the use of facets. Basically, the shadow provides the meta-information about the facet. It is a text section of semantic description of the facet. It will be represented in a human and machine readable form using the OTDL (Ontology-based Task Description Language) and thus can be accessed by developers, users and machines alike. The OTDL is extended from W3C's OWL.
- **Code Segment.** This is the body of the executable code which implements the functionality. To ensure portability, the code segment will be written in Java. The code segment may contain several Java classes, but only one of them contains the publicly-callable method corresponding to the functionality contract. Functionality adaptation involves changing the way an application carries out its functionality. Particularly, it involves changing the execution of the application: (1) using different sequence of actions or different algorithms, (2) using code enhanced for certain platforms, (3) tradeoff between memory space and execution time efficiency (i.e. using less memory but more processing time to carry out a task), or (4) partitioning the task so that it is partially executed on a server or a nearby peer device rather than completely locally. The workflow of how we achieve the functionality adaptation is explained in the next section.

Intuitively, the facet is like a function called by name and the runtime constraints. The matching of the name and runtime constraints of the function provides the adaptivity. Facets are hosted on facet servers. A facet can be discarded from the run-time as soon as it is used. If it is required again, another compatible facet can be brought in from the network and used. Clients request for facets from the proxy servers, which recommend suitable facets to clients taking into account their device configuration, the surrounding execution environment and the user preferences. Because facets are brought in at run-time to compose the required application, applications can dynamically adapt to the client devices and be made context-aware. If there were two facets of same functionality, the facet which is more suitable to the client device would be brought in. With such mobile and adaptive features, users can perform the same functionality on different devices wherever they go.

In our system, a facet is not a stand-alone application. Every application is associated with a container. It contains a UI which interacts with the user and a set of functionalities that the application can offer. These functionalities are stored in the container as facet specifications which include funcIDs. The UI provides a means for users to access the functionalities offered by the container. When a particular functionality is required, the corresponding facet is brought in from the facet servers according to its specification from the facet servers, in turn starting off the execution of a whole tree of facets. The container also keeps track of state information so that we can restore execution when it moves to another device.

Figure 3 shows an example application of using facets in Java. A facet is invoked by filling in a *FacetRequest* with specifications, initializing a Facet object and invoking *execute* on it.

3.3. State-on-demand execution scheme

In our system, the LMCS migrates a mobile agent by chopping the execution state into segments and pushing the top segment to the remote site while keeping the residual

```
// Filling in a facet request
FacetRequest fr = new FacetRequest();

fr.addCriteria( functionality_id ,  200007 );
fr.addCriteria( vendor ,  SRG.CSIS.HKU );

// Initializing a facet
FacetInterface fBlur = (FacetInterface) new Facet(this, fr, fcontainer);

// Invoking a facet
Object[] input = null, output = null;   output = fBlur.execute(input);
```

Figure 3. Program using facets.

state at the local site. The target site, upon receiving the segment, will try to restore the partial stack with the bottom frame pointing to a handler for fetching the next frame segment when the current segment is consumed. The size of the execution stack of the running agent is therefore kept small during execution, which results in memory saving.

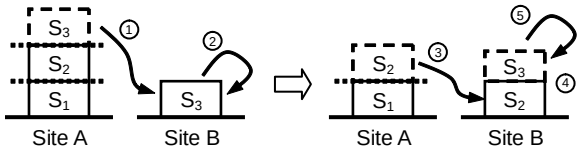

Figure 4. An illustration of the SOD mechanism.

Figure 4 illustrates the SOD scheme. The frames are chopped into three segments. The last segment S_3 is first migrated to the destination site and executed. Just before S_3 finishes executing (i.e., hitting the return point of a method), the instrumented code of S_3 checks to see if it is the bottommost frame in the transferred segment. If so, it throws an `ReachBottomException`, and its execution state is again captured and stored into a frame S_3'. Upon being notified of the exception, the LMCS contacts the site where the following segment S_2 resides, and retrieves the segment and inserts it before S_3'. The execution of S_2 then restarts. The whole process repeats until all available execution segments finish their execution.

The implementation of SOD relies on a data structure called *StackFrame*, which is an abstract object used in the instrumented code to assemble the real Java frames in the execution stack. When migration is triggered, a `NotifyGone` exception would be raised. An exception handler in the instrumented code then puts the values of the local variables and the PC into a *StackFrame*, and appends the *StackFrame* to a stack list. The process is repeated until all the frames are captured. After that, the list of *StackFrame*s is segmented according to the resource requirement of the target site. The segments are then serialized and sent to the target.

When a frame segment is received at the destination MCS, the state of the execution stack is re-established by invoking a *restore* method in the bottommost *StackFrame* object of the segment. Execution states stored in *StackFrame* objects of the segments (and thus the execution) are then restored sequentially with the properly-instrumented codes. At this stage, facets are lazily fetched and loaded to avoid bringing in unnecessary or already-cached code, thereby reducing redundant bandwidth consumption. After some execution and when all the available frames (and their corresponding methods) in the segment are exhausted, the underlying agent system would contact its previous hosting agent system to ask for the next segment, and the process continues until all segments are consumed.

The amount of bandwidth saved by our SOD scheme is best expressed by the concept of *lazy-frame*. We define *lazy frames* as the frames (or objects) that are not migrated after two or more transfers under the SOD scheme. A bottom frame is always lazier than a top frame. If the bottom frame segment is only needed at the N^{th} hop during the execution of the agent, the bandwidth of transferring the segment will be saved in the first $(N-1)^{th}$ hops. Under this definition, the percentage of total bandwidth saved can be roughly expressed as the ratio of the total size of lazy frames to the total size of frames that should be migrated by the normal mobile agents, because lazy frames are the frames that would be migrated in normal cases but not in our SOD scheme. Therefore, increasing the lazy-frame-to-non-lazy-frame ratio can help to optimize bandwidth usage with our SOD scheme. In addition, the use of *non-uniform segmentation scheme*, where the potential non-lazy frames are put into larger segments, can further reduce the overhead of the segmentation.

4. Evaluation of our system

The resulting system was tested on two standard PCs equipped with an Intel Pentium III 650MHz processor and 128M memory running Sun JVM on Linux 2.4.18 with JIT compilation enabled. We used three simple recursive applications, which exhibit different execution behaviours, to evaluate our SOD: the Fibonacci program (`Fib`), the quicksort program (`QSort`), and the N-Queen program (`NQueen`). We conducted two experiments to analyze the memory and bandwidth consumption of our scheme when the agent performed single-hop or multi-hop migrations. The results are evaluated based on the overall behavior of our SOD scheme.

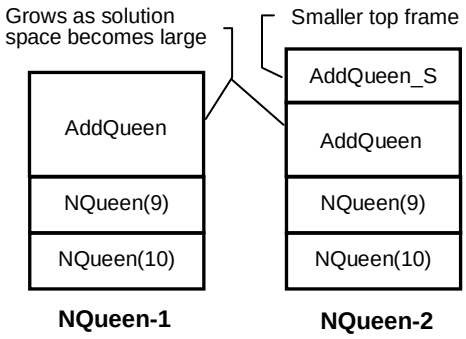

Figure 5. The status of runtime stack of `NQueen` application

4.1. Migration under single hop scenario

In the first experiment, we aimed at a general idea of how memory is consumed when an agent migrates from one host to another (the single-hop scenario). We used the `Fib` program in this experiment. With the SOD scheme, a stack of 17 frames formed by executing `Fib(35)` was uniformly segmented, so that each of the resulting segments contained two stack frames. The experiment result shows that with SOD, we only needed to use 529 bytes of memory on average to migrate the execution, which was much lower than that needed in a typical execution without SOD (which required 2211 bytes). This shows that SOD allows a migration to be done in a memory-limited environment. However, because there is certain overhead in transferring a segment, bandwidth was not saved in this single-hop scenario. Nevertheless, the overhead can be balanced out by the bandwidth saved when the LMA hopped across multiple hosts, as illustrated by the results in the next experiment.

4.2. Migration under multi-hop scenario

In the second experiment, we evaluated the effectiveness of the SOD scheme in terms of memory and bandwidth saved by arranging the three testing applications to have different traveling patterns across multiple sites (the multi-hop scenario). We used two different ways to implement the `NQueen` application to visualize the impact of different execution behaviors with our SOD scheme. The first one (`NQueen-1`) was implemented in the normal recursive manner, while the second one (`NQueen-2`) was implemented with some optimization using the lazy frame concept (Figure 5). The function ADDQUEEN in `NQueen-1`, which contained a large iterative structure to generate the solution set, was replaced by two functions ADDQUEEN_S and ADDQUEEN' in `NQueen-2`. ADDQUEEN_S had a smaller iterative structure, and ADDQUEEN' contained an additional loop to iteratively call ADDQUEEN_S. With this modification, although the two functions behave the same as ADDQUEEN, the top frame for `NQueen-2` (ADDQUEEN_S) became smaller, as a smaller solution set was generated by a smaller iterative structure. Also, it was still fair to make comparisons between `NQueen-1` and `NQueen-2`, because in `NQueen-2` only the program structure on how to manipulate the available states was changed, and that did not increase the total size of execution states.

The setup of the experiment was simple: agents (which carried the testing applications) were forced to migrate every 15 seconds after they arrived at a host. For agents that adopted the SOD scheme, non-uniform segmentation was applied to the execution stack: more frames were put inside a segment toward the top of the stack so as to reduce the overheads that were caused by excessive transfers. The corresponding number of hops, bandwidth and memory usage of the agents in the experiments were shown in Table 1. It was found that while `Fib` and `NQueen-1` obtained relatively small bandwidth gain (3.64% and 4.22% respectively), `QSort` and `NQueen-2` could get high bandwidth gain (66.5% and 51.8% respectively).

Besides bandwidth saving, all the above experiments illustrated the low memory usage under the SOD scheme. Depending on the execution behavior of the application and the segmentation scheme, we needed on average only 21% of memory for SOD agents to carry the applications when compared with normal agents (who have to carry all the execution states). This suggests that applications can be run using little memory under SOD, making them more favorable in the pervasive computing environment.

4.3. Discussions

The experimental results can be best explained using the lazy frame concept. For `Fib`, the frames are small and similar in size. Since only simple computations are involved in each recursive call, the frames on the stack are quickly consumed. Also, due to the recursive fan-out, top frames are consumed at a faster rate than bottom frames. All these indicated that many non-lazy top frames were created (~ 25 during each agent transfer). Therefore, although we saved the bandwidth of 3 to 4 lazy frames per agent transfer, the saving was insignificant when there were frequent transfers of non-lazy frames. Adding the effects of overheads introduced in segmenting the frames, the percentage of bandwidth saved per site for `Fib` was small, and so was the total bandwidth saved.

The case for the `QSort` program is totally different. For `QSort`, the frame size grows exponentially (roughly) due to its divide-and-conquer algorithm, but at the same time the frames are consumed at a much slower speed than `Fib`

Testing application	Fib (35)	QSort (5000)	NQueen-1 (10)	NQueen-2 (10)
Number of agent hops	12	12	40	40
Total bandwidth used (normal)	327088	1715062	33600410	33932825
Total bandwidth used (with SOD)	315197	574655	32183569	16365801
Total % bandwidth saved by SOD	3.64%	66.5%	4.22%	51.8%
Avg. normal agent size	27257	142922	767997	848320
Avg. initial mem. usage with SOD	5974	15773	685827	259733

because of the iterations on swapping values and the effect of recursive fanout. These execution behaviours altogether led to creation of large lazy frames, thereby diluting the impact of the overheads introduced by non-lazy frames, and yielding a high percentage of bandwidth saved.

The concept of lazy frame can be further reinforced by the `NQueen` experiment. In executing `NQueen-1` (normal implementation of `NQueen`), a large amount of computation time is spent on an iteration in the topmost frame ADDQUEEN (finding a safe position for queen), where a large execution state (solution sets) resides. As there is no recursive fanout and no computation prior to the recursive call, the small bottom (lazy) frames therefore do not contribute much to bandwidth saving. Comparatively, although `NQueen-2` exhibits similar facet-invoking behavior, its top frame, ADDQUEEN_S, has smaller execution states (solution sets). Also, since ADDQUEEN', which invoked ADDQUEEN_S, accumulates the solution sets returned from ADDQUEEN_S and the time for executing ADDQUEEN_S is long, larger lazy frames and smaller non-lazy frames are produced, which results in significant bandwidth savings.

From the above discussions, we find that traditional agent applications, which traverse multiple hops by the agent itinerary, can benefit from SOD in terms of memory and bandwidth reduction. One may then think that SOD is not beneficial to agents that make relatively few hops, such as agents that carry end-user applications. In fact, for these applications, SOD can still give reasonable reduction on bandwidth usage. In addition, SOD can help to cut down their memory footprint, thereby making it possible to run these applications in small-memory devices.

5. Related work

There are currently many MAS implementations available (a good survey of mobile agent systems can be found in [4]). Since the use of mobile agents has to address the issue of platform heterogeneity, these systems are usually implemented using platform-independent programming languages. We can classify them into two types. One type is implemented based on some research interpreted languages like Tcl, Scheme, Oblique and Rosette [5]. Example of these MASs include Telescript by General Magic, the first commercial mobile agent platform as it claims to be, Agent TCL, TACOMA and ARA, etc. Since the time when Java was launched, dozens of Java based MAS have been developed, which represent the second type of MAS. In fact, the class-loading feature of the Java Virtual Machine, coupled with techniques such as object serialization, RMI, multithreading and reflection, have made the building of MASs much simpler.

Aglet is a commercial MAS developed by Tokyo Research Laboratory of IBM, which was made open-source recently. It is one of the most widely used MASs for developing mobile agent applications, and has definitely received the most coverage in the literature [5]. It uses typical Java features such as object serialization and RMI, and provides a graphical Aglet control interface called Tahiti [7]. It supports only weak mobility, which implies that in order to deploy the full power of the mobile agents, programmers need to maintain the relevant execution states themselves, which is error-prone and makes the resulting code not maintainable. Our mobile code system has a similar architecture to that of Aglet. But our system supports strong mobility, so that the agent can migrate at any point during its execution. This allows the power of the mobile agent to be fully exploited. Since Aglet does not seem to have a plan to work in the pervasive computing environment, it has not focused on making the mobile agents lightweight.

James MAS [11] is a research-based MAS that was co-developed by the University of Coimbra and Siemens, and was mainly oriented for telecommunications and network management, with performance being an important issue. They suggested (but not implemented) that a special type of agents called *elastic agents* could be used: these agents can drop some code when they do not need the code, or load some new code when needed. This approach is similar to our design of LMA. But our scheme pushes the idea one step further: instead of using the code components, we use our facets which themselves can be adaptive to the environment. The overall network bandwidth can therefore be further reduced.

The Nomads MAS [13] supports strong mobility of mobile agent through using the Aroma VM. The Aroma VM implemented a state-capturing mechanism in the JVM kernel. Instead of using the VM modification method to implement strong mobility, our mobile code system uses the source code instrumentation method via a preprocessor or compiler that adds state-saving code inside the agent code. The main benefit of our approach is that the resulting agent

programs can be readily executed on heterogeneous platforms having the standard JVM.

6. Conclusion

In this paper, we describe ways to provide lightweight and flexible computations to the pervasive computing world. To reduce the usage of memory and network bandwidth, which are the two most valuable resources in a pervasive computing environment, we have devised the State-On-Demand (SOD) scheme to bring in code and associate execution states of an application dynamically after a migration. The effect is that the scheme enhances the flexibility and adaptability of the system.

To prove our concepts, we have implemented a simple working prototype for our mobile code system in Java, which incorporates some instrumentation strategies for supporting the SOD execution scheme. The experiments have shown that SOD can in fact help reduce the bandwidth and resource usage in the target system through the use of facets and LMAs. However, further research is needed to create quantitative models to evaluate and strategies to optimize the bandwidth and resource consumption in a pervasive computing environment.

We believe that our work has made an initial yet critical step in providing a mobile computing framework for pervasive computing environments. It lays a foundation for the creation of lightweight and flexible computations that meet the need of mobile and nomadic users. Our research shares some common goals with other hot related research areas, including grid computing and peer-to-peer computing.

References

[1] C. Bäumer, M. Breugst, S. Choy, and T. Magedanz. Grasshopper: a universal agent platform based on OMG MASIF and FIPA standards. Technical report, IKV++ GmbH, 2000.

[2] Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Communication of the ACM*, 42(3):88–89, March 1999.

[3] G. Vitaglione, F. Quarta, and E. Cortese. Scalability and Performance of JADE Message Transport System. In *AAMAS Workshop*, Bologna, 2002.

[4] Fritz Hohl. The Mobile Agent List. URL `http://mole.informatik.uni-stuttgart.de/mal/mal.html`.

[5] Joseph Kiniry and Daniel Zimmerman. Special Feature: A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4), July 1997.

[6] Krishna Akella and Akio Yamashita. Application Framework for e-business: Pervasive computing. URL `http://www-106.ibm.com/developerworks/library/pvc/index.html`.

[7] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[8] Niranjan Suri, Marco Carvalho, Robert Bradshaw, and Jeffrey M. Bradshaw. Small Mobile Agent Platforms. In *Workshop on Autonomous agents and Multiagent systems*, Bologna, 2002.

[9] Robert Grimm, Janet Davis, Eric Lemar, Adam MacBeth, Steven Swanson, Tom Anderson, Brian Bershad, Gaetano Borriello, Steven Gribble, and David Wetherall. Programming for pervasive computing environments. Technical Report UW-CSE 01-06-01, University of Washington, Department of Computer Science and Engineering, Seattle, June 2001.

[10] Tatsurou Sekiguchi. *A Study on Mobile Language Systems*. PhD thesis, The University of Tokyo, 1999.

[11] L. Silva, P. Simões, G. Soares, P. Martins, V. Batista, C. Renato, L. Almeida, and N. Stohr. JAMES: A Platform of Mobile Agents for the Management of Telecommunication Networks. In *3rd International Workshop on Intelligent Agents for Telecommunication Applications*, Stockholm, Sweden, August 1999.

[12] Object Space. Voyager core package technical overview. Technical report, Object Space, March 1997.

[13] N. Suri, J. Bradshaw, M. Breedy, P. Groth, G. Hill, and R. Jeffers. Strong Mobility and Fine-Grained Resource Control in NOMADS. In *Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents, ASA/MA 2000*, pages 2–15, Zurich, Switzerland, September 2000. Springer-Verlag.