Chapter 1

Introduction

1.1 The new needs in Pervasive Computing

Pervasive computing is undoubtedly the next wave in the computing industry. With the proliferation of popularity of Intelligent Appliances (IA), e.g. mobile phones and Personal Digital Assistances (PDAs), and wireless network technologies in the past few years, pervasive computing has successfully gained wide awareness in the areas for system design. However, because of the inherent limitations on battery power and hence resources, application developers may sometimes find it difficult to develop software on these IAs. In view of this, there is a need of designing an architecture that enables computing in small on these small devices.

On the other hand, as ubiquitous devices become more popular, there are more and more data being generated every day, e.g. telephone number, meeting schedule, email addresses, stock quotes, or even voice memo and video conferencing record. These data usually exhibit different natures: some of them are private (e.g. email address), some of them are widely distributed over the network (e.g. stock quotes), whereas some of them are too large to be transfer in a bandwidth limited environment (e.g. video conferencing record). Despite this, in the current computing world, the way in manipulating these data usually only involve the use of a static server hosting the required service. In other words, different strategies must be applied to move the data from its source to its dedicated server. Many problems like security, privacy, high bandwidth usage, etc. will therefore arise. Clearly, this *client-server* approach only complicates the interaction between different entities in this scenario, and makes the resulting system error-prone.

With only one computational model being used in the whole inter-connection network system is clearly inadequate to satisfy all the users needs. It is because data that are large in size or private in nature, as those in the examples presented above, are not supposed to be moved away from their home site. Furthermore, it may be time and bandwidth consuming to gather data that is widely distributed over the network. In view of this, people demand a *flexible* computational model in managing, exchanging and processing data. They need an execution infrastructure supporting different modes of computation, thereby enabling *nomadic* and *convenient* data access through their handheld devices. Also, with such a flexible execution infrastructure, features like *peer-to-peer computing* or *ad-hoc networking*, which are the technologies that are actively discussed nowadays but still in its infancy stage in the area of mobile computing, can now be made possible. As one may imagine, enabling such kind of technology would be valuable in making communications and information sharing among different pervasive users much easier, automatic and secure.

Mobility is the key to generalize the degree of flexibility that users need. We here identify three main categories of mobility issue: device mobility, user mobility and service mobility. ***Device mobility*** enables mobile devices to receive continuous access to services independent of their current location [10]. This allows users to access their personal information and enjoy the services over the interconnection network by simply carrying a physical device along with them. It is the issue that is usually addressed by the use of ubiquitous devices and wireless technologies with mobile networks. Another mobility issue, which is even more important, is known as user mobility. ***User mobility*** allows a user to access and manipulate his personal data with the same settings on any devices. For example, with user mobility, people can walk in any cyber cafes and access the PCs there with their customized application interfaces and settings, e.g. wall papers, icons location, etc. These user settings can be stored somewhere on the network, or even on PDAs. Even more, with user mobility, the users are now able to access the data (e.g. documents and some personal settings) on the PDAs and on PCs interchangeably, virtually removing the device boundary.

On the other hand, while user mobility take care of the issues on data movements among devices for the end users, the application or service used to manipulate their data are still static from the end users' point of view. This is not a problem if the service they use is hosted on the local device. However, if the service they use to manipulate their data resides on a remote machine, when the user moves away from the location where the service resides, the service accessing latency would be increased. It is clearly unfavourable when the user needs to send large amount of data to the remote machine for processing. Even worse, the network is highly unstable and dynamic in the pervasive computing environment. Users therefore may not be able to reach the static services hosted on services when this disconnected conditions occur. ***Service mobility*** therefore comes in and addresses these issues by giving services the capability to move to the nearby hosting sites, so that the users can access the services irrespective to the current time and their current location. By this, a system featuring service mobility can also give an illusion to the users that the required service would "follow" them wherever they move without suffering the problems of increasing network latencies and disconnections.

While device mobility is quite well addressed by the current hardware technologies and mobile network infrastructures, designs and solutions targeting user and service mobility in the pervasive computing environment are rare. Conventional approach in addressing the user mobility issues is to adopt the centralized server approach. As in many common thin-client systems, a central server is used to store all or part of the application data of the user, so that the users can access these data on different service end-points at different locations on different devices. However, such a server intermediation approach may not be suitable in the pervasive computing environment. It is mainly due to the *data privacy* issues thereby arisen and the inherently *volatile low-bandwidth* nature of the wireless network in the pervasive computing environment.

Nevertheless, the absence of server creates another new sets of problem in the pervasive computing world. In the case where the server is absent, applications have to be designed in such a way that is suitable to be executed on all end-users' devices. This is especially difficult in the pervasive computing environment, which is highly heterogeneous and a wide range of devices participate in it. In that case, applications need to have many different implementations even if they just need to provide similar services to end-users that want to compute on different computing platforms. To address this issue, *service mobility* is the key. With service mobility and its corresponding adaptation ability, services can now move and adapt to the client device, getting aware of the changes in client-side environment and making the according change in terms of structure and resource usage of the application. At the same time, a decentralized design on addressing user mobility is achieved. The problems arisen from centralized control can therefore be avoided.

To sum up, we have identified three main requirements for a system designed for pervasive computing:

- **Lightweight.** The execution hosting on the system must be *small* in resource and network bandwidth usage, so that it can work even on the small mobile devices with limited bandwidth and unstable network connections.

- **Flexible.** A lot of information and data will be generated and flying about the network in the pervasive computing environment. The different natures of data and the heterogeneity of the environment necessitates a flexible computation model in managing, exchanging and processing data. The system design should therefore provide different kinds of mobility support to allow flexible interactions between end-users, devices, data and services.

1.2 Our approach to address the needs

It is clear that a computing model that enables *small* and *flexible* computation is the key to success in the pervasive computing environment. In fact, the introduction of the concept on mobile codes nicely answers the need on providing flexibility to the system. Three mobile code paradigms have been identified so far: *Code On Demand* (COD), *Remote Evaluation* (RE) and *Mobile Agents* (MA) [6]. They can nicely support various kinds of mobility through the proper combinations in moving the codes and states. Moreover, the use of them naturally enables two important features in pervasive computing:

- **Peer-to-peer computing.** Peer-to-peer computations can be enabled easily by simply allowing the mobile codes to actively navigate and execute on the authorized peer devices. The emergence of peer-to-peer computing technologies is drawing more attention in the pervasive computing world, mainly because its highly dynamic nature perfectly fits the computation infrastructure in the mobile environment. While traditional centralized client-server model easily reaches performance bottleneck and experiences availability problems, the problems can be easily addressed by the *distributive* and *ad-hoc setting* nature of the peer-to-peer computing infrastructure (imagine that we can send mobile codes to some available and idle servers to execute services that are customized for us). On top of that, in the client-server approach, since all the computations must be carried out on the server-side machines, if the computation involved requires the usage of some private data, a trust relationship is forced to be established between the client and server, or otherwise the computation cannot be carried out. However, through the use of mobile code in the peer-to-peer architecture to achieve *service customization*[1], the user can now freely choose a peer site that he trusts before he sends the mobile codes to manipulate his private data.

- **Disconnected operation.** As mentioned in the previous section, one of the several unique

[1]Service customization allows service requester to execute their own codes on any computing platform, thereby making an illustion that the providing service is customized. More would be discussed in Section 2.1

characteristics of the pervasive computing environment is its network instability. Networks in this environment are usually considered to be unreliably connected when compared to the wired networks. There is always a saying in the field of pervasive computing: “Disconnection is the rule, not the exception” [14]. Disconnected operation is therefore a valuable feature in this environment. With the use of mobile code, this feature can be easily enabled by running the required mobile code on a host that is reachable by the user device.

On the other hand, since disconnected access make mobile users more difficult to support and manage, the system design should also be *intuitive* and *automatic* to most users, so that least efforts on supporting and maintaining the system are needed. In this case, the common technologies on mobile agent, a form of mobile code, offers the intelligence and intuitiveness that is required in user managements. For example, the user can instruct the mobile agents to find and explore the information they need over the network. The intelligence of the mobile agent can then drive itself to search for the relevant information without interacting with users, even if the user disconnects from the network. This essentially addresses the problem.

In addition to these two features, the use of mobile codes can give extra benefits to the mobile computing environment that cannot be achieved by the traditional client-server computing model, e.g. the service customization feature described above. However, we notice that applying one mobile code paradigm *alone* may still not be flexible enough to deal with pervasive users’ needs and resource requirement. For example, conventional mobile agent technologies usually carry *all the required codes* together with them along their itinerary. Since the codes they carry would not change nor be “dropped” after execution, the agents may not be lightweight enough in terms of bandwidth and resource usage. Also, the agents may not be flexible enough to cope with the rapid environment changes when they hop between different devices in the pervasive computing environment. Even worse, the approch of carrying all the codes is clearly not *lightweight* enough to satisfy the need in the pervasive computing environment with limited network bandwidth and computing resources.

The Sparkle architecture therefore comes into the picture. The Sparkle architecture aims to offer a design that enables different kinds of applications to be executed over all ranges of devices in the pervasive computing world. In other words, lightweight executions are needed to be enabled on some mobile devices that are small in resource and bandwidth usage. To achieve this, our architecture essentially separate the manipulation of functionalities and its required data. The functionality can therefore be broken down into many small pieces of functionality components, which we call it *facets*. These facets are in fact the code components that exhibits two important features: (1) each facet carries out single functionality according to its contract, and (2) it has no residual state. In that sense, the formation of a service can simply be done by grouping all the required facets agreed on certain contracts describing the functionalities needed. There is no absolute and permanent relationship between two facets: any one of them can be plugged in and thrown away dynamically during execution. Lightweight executions can therefore be achieved by choosing the facet that is suitable to be executed in its target execution platform. This facet adaptation task is managed by the Intelligent Proxy in our architecture.

With only the introduction of facets and Intelligent Proxy, the flexibility of mobile computations cannot be exploited in the pervasive computing world. This is where two main components in our architecture: the Lightweight Mobile Agent (LMA) and the Lightweight Mobile Code System (LMCS), comes in and answers the need. Briefly speaking, the LMAs allow the required facets to be dynamically plugged in and removed. By doing so, the codes required to achieve its carried goal can be changed dynamically at run-time to cope with the environment changes along its itinerary. This model is more flexible and natural than the conventional approach of implementing mobile agent described above.

Besides the additional computation flexibility it offers, the use of LMA also enables *lightweight* computations. In order to delegate the execution of a particular task, instead of putting all the required facets into the LMA, only the functionality descriptions of the facets are needed to be put into it. Of course, the corresponding information about the execution status should also be carried along with it. When the LMA has traveled to its destination site, it would recover the execution status and load in the required facets. Since facets can now be brought in and loaded dynamically from the nearby Intelligent Proxy, there is no need to bring in all the codes at one time. The *laziness* in bringing in codes on demand, which thereby reducing the movement of code, helps to reduce the overall bandwidth requirement. Also, since the size of code being downloaded can be adapted to the new environment, the bandwidth requirement in transfering the required facet can be reduced if the target device has limited resources. On the other hand, we have also found a similar on-demand scheme to further reduce the bandwidth usage of transfering the execution states and the resource needed to resume the migrated execution. This is done by chopping the frames on the execution stack into state segments and be on-demand transfered to the remote site segment by segment. Through using these two on-demand schemes, the execution can be made lightweight.

Since the mechanism in reducing the memory and bandwidth usage involve the use of two code mobility paradigms: the MA paradigm and the COD paradigm, we think that it is improper to name the underlying agent-hosting system as a *Mobile Agent System*. As more different forms of code mobility is involved on our system, we name it as a *Mobile Code System* instead.

1.3 On strong mobility support

Although we know from above that the use of mobile agent computing model and its relevant technologies is generally beneficial to the pervasive computing world, the real flexibility that the mobile agent paradigm provides is rarely fully exploited. It is mainly because most of the current MASs are implemented using Java programming language. Java, as a programming language that is commonly used in deploying network applications in recent years, exhibits features that are favourable in developing and supporting mobile agent technologies. Its *use of virtual machine technologies* and the support of *dynamic class loading* features make it simpler to deploy and manage the mobile agents in the highly heterogeneous pervasive computing world. However, there is one major drawback in supporting mobile agent technologies in Java-based MASs: *Java forbids the dynamic inspection of the execution status*, including the stack status, the value of the program counter, and the status of the threads that are executing. In other words, these execution status are not able to be transfered nor recovered once the mobile agents migrate to another site.

Some traditional MASs simply ignore this problem. They mandate every mobile agent program to start executing at a predefined point when the mobile agent migrates, and leave the responsibility of maintaining the relevant states to the agent programmers. We argue that doing so will greatly decrease the flexibility of the resulting model and also increase the average code size of the mobile agents in order to deal with different situations in the pervasive computing environment. These MASs are usually classified as *weak* and supports only the so-called *weak mobility*, in which no execution status are transfered to the remote site. Real mobile agents should, on the other hand, work on MASs that support *strong mobility*, which allows executions to be recovered at the same point and state even after migration. Only then, the original nature of the mobile agents of freely hopping from site to site can be realized, and the power of it can be fully exploited.

In recent years, active researches have been conducted on simulating strong mobility on weak MASs. Typical approaches on doing so include instrumenting the codes at different levels and modifying the underlying system constructs (e.g. Java virtual machine). However, while the former approach give solutions only on extracting and migrating the status of the execution stack and its associate program counter, the latter approach vanishes the portability feature inherited from Java. Therefore, in the current literature, there seems to be little support on *portably* migrating the status of inter-thread communications, e.g. interlocking states, waiting states, etc. This *thread-state migrating* feature is believed to be useful especially when the mobile agents want to deploy and execute several tasks concurrently over the target execution platform. In view of this, we have studied and found out ways to support portable migration of thread status. We have also proved our concept by modifying a source-code preprocessing engine, with our strategy on supporting thread-state migration added in. We believe that using this engine, in cooperation with the LMA and LMCS, can make the power of the mobile agents fully exploited in the pervasive computing world.

1.4 Thesis Contribution

To our best knowledge, LMCS is the first mobile agent platform targeted at the pervasive computing environment that aims to provide a working environment for the mobile agents over a large range of devices. It helps to bring the benefits of mobile agent technologies to the pervasive computing world. The contribution of our work is listed as follows:

1. LMCS, which works in cooperation with LMA, provides a general mobile agent framework that is suitable to be run in the pervasive computing environment. Through incorporating the flexibility provided by the mobile agent technologies into the pervasive computing environment, our system has essentially addressed the issues of providing user and service mobility. It has essentially offered a wide range of new possibilities to the application designs for pervasive computing, including peer-to-peer computing applications and disconnected operations.

2. Strong mobility is an indispensible features to enable the computation flexibility provided by the mobile agents. In this research, we have also found out ways to strengthen the support of strong mobility on weak mobility system. Through using the source code instrumentation scheme designed by us and our source code preprocessing engine that adopts the instrumentation scheme, the mobile agents hosting on the LMCS are allowed to strongly migrate even when they are executing parallel tasks using multi-threading techniques. To our best knowledge, it is the first MAS that can *portably* support the strong migration of thread states over weak mobility systems, because our scheme does not need the modification of the Java virtual

machine. On the other hand, the instrumentation schemes designed by us is language neutral. It can be readily applied to other programming languages that adopts a thread execution model similar to that in Java, e.g. C#.

3. In view of the large size of the traditional mobile agents, which grows even larger after instrumenting the relevant codes to support strong mobility, we have also provide an on-demand schemes on bringing in the codes and its associate execution states, in order to make the resulting agent program lightweight upon migration in terms of both memory resources and network bandwidth usage. The underlying principle of this is also language neutral.

4. By allowing the mobile agents to travel also on the small mobile devices, it enables *peer-to-peer computing* in the pervasive computing world. It essentially puts a step-forward to the ultimate goal of Grid Computing: in which the users will be insulated from the complexity of the underlying systems, enabling them to easily access the global resources (both computational resources and data) that they need to get their work done.

In other words, my thesis contributes mainly on the mobility aspects of the mobile agent technologies and how the computation flexibility of our system can be enhanced. My thesis does not strongly address other aspects about the mobile agent technologies such as security issues, agent communication issues and standardization issues, although schemes from other researches may be readily plugged into our architecture.

1.5 Thesis Organization

The whole thesis is mainly divided into nine chapters. Chapter 1 is the introduction. Chapter 2 gives a brief review on the literature on mobile agent systems and its related technologies. Chapter 3 briefly introduces the whole Sparkle architecture and the role of LMCS and LMA inside it. Chapter 4 introduces backgrounds, terms and concepts related to code mobility and states managements. Chapter 5 gives our code instrumentation strategy on portably supporting transparent migration of the multi-threaded programs (i.e. strong mobility) over weak mobility systems in detail. Chapter 6 describes our strategies on making the agent execution and migration lightweight. Chapter 7 discusses the architectural design and implementation issues of our whole mobility system. Chapter 8 analyses our approaches through some experiments and some evaluations and discussions on our scheme would be given. Finally, Chapter 9 concludes our works and describes the possible future works on our project.

Chapter 2

Literature Review

As we have seen from the previous chapter, the introduction of mobile agent and mobile code technologies have undoubtedly created opportunities for increasing the computation flexibility in the pervasive computing environment, making the applications more adaptive to the ever-changing user and environmental needs. Therefore, in this chapter, we would briefly explore the current literature related to mobile codes and mobile agents (more focus would be put on the current mobile agent technologies, which are under active research nowadays).

Using web services [40] is by far the most common approach to address the problem of performing computation for small devices, and it is another hot topic in the field of computer science nowadays. However, since it adopts the traditional client-server computation model, it has some short-comings in providing enough flexibility to application developers for fulfilling pervasive users' needs. However, the use of mobile agents and mobile codes can complement this area of users' need. In Section 2.1, we would look at the benefits of using the mobile agent and mobile code technologies over that using the traditional client-server approach, and see how its flexibility can compensate the pervasive users' needs.

As mentioned above, the community on mobile agent researches has grown to be active in recent years. It opens up many new areas of researches, including knowledge representations between agents, the portability and standardization of mobile agent technologies, the agent securities, the control strategies of mobile agents, etc. In Section 2.2 and Section 2.3, we would briefly describe the characteristics of three mobile agent systems, and also the major areas of current mobile agent researches.

After that, in Section 2.4 and Section 2.5, we would study how the flexibility provided the mobile agent technologies can help the growth of yet another two active research areas: peer-to-peer computing and grid computing. We will explore the relationship between mobile agents and them, and what a difference will be made when the mobile agents is introduced in such area.

Finally, in Section 2.6, we will discuss the common approaches used in addressing one of our major goal: *user mobility*. In particular, we will study a system, the one.world system, that aims to achieve this goal, and compare their approach and design principles with our mobile code system.

2.1 Computation flexibility introduced by mobile codes

Pervasive computing has undoubtedly created a new set of computing requirements that cannot be fully satisfied by the traditional computing models like the Client-Server approach. The design of LMA, with some of the common properties of the mobile agents, overcomes a certain number of them using its computational flexibility. In this section, we will explore what kind of new computing model can be introduced by incorporating the mobile agent technologies (equipped by some of the Code-On-Demand schemes being used by LMA) into the pervasive computing environment. In reading the following subsections, one may take the popular technology nowadays, the Web service, as a typical example of the client-server computing model, and will find that how mobile agent technologies can help to increase the flexibility of the current computing models.

2.1.1 Disconnected operation

Unstable network connections is believed to be the rule in pervasive computing environment [14]. However, some traditional computation models, like the client-server computing model, do not quite well-address this issue: in the client-server approach, all the computations has to be done by the services resided on the server side. If a user, who wants to use a particular service, cannot connect to the corresponding server because of the partial failure of the network, he simply cannot carry out any computations with respect to that required functionality.

With the introduction of the mobile agent technologies coupled with the Code-On-Demand scheme (i.e. the principle of our LMA design), the user can still perform some limited computations through using a decentralized approach. In our design, since code components are needed to be downloaded to the client device, the client device may have a chance on caching the code components. In this case, if the device has some code components that are required by the computation in the cache, these components can be directly retrieved from the cache and used to carry out some computation even if the device is disconnected from the server. Also, even if the code components required by the computation is not cached, if the client device is still connected to a closed network (i.e. a network that cannot reach the server, but is still connected to some other peer devices), it can send out a mobile agent to navigate the other peer devices on the network to see if the required code components are available. If it can be found, the agent can bring it back to the client device. This can be done easily by setting the mission of the mobile agent sent to be "to find the required code components", in which we treat the code component as a computing result.

Through using this decentralized *peer-to-peer* collaboration approach, the *single point of failure* hazard introduced by the centralized client-server approach can be eliminated.

2.1.2 Asynchronous computation

Another benefit in using mobile agents in the pervasive computing environment is its provision of asynchronous computation support. After sending out an mobile agent to perform a pre-designated task, there is no need for the client device to maintain the connection with agent. Instead, through using some *agent tracking* techniques in the MASs, the end-user can *retract* the mobile agent whenever they want after detecting that the agent has completed its designated task. In this case, since the computation does not require the connection between the client and the computation server to be permanently established, the problem of the unstable network connections can be overcome: the end-user can now get the computation result whenever they get re-connected to the server.

2.1.3 On providing user mobility

User mobility enables a user to continue a task as he changes his location or device. To achieve user mobility, typical client-server approach would make use of a central server storing the information about the execution status of a migrating computation. When the computation moves to another platform, it just needs to retrieve the corresponding execution status from the central server, and recover the execution from it. This approach, however, may not be suitable in the pervasive computing world, in which a central server is usually not available or expensive to be reached because of the limited connectivity in the mobile computing environment.

With the mobile agent technologies, the information about the current execution status can now be directly transferred to the destination system by using the built-in status migration mechanism. Together with the code components that are brought in on demand and adapted to the user environment, the execution of the user task can be continued on whichever device and wherever he is.

2.1.4 Peer-to-peer computation

For a system adopting the mobile agent technologies, unlike the centralized client-server approach, no intermediate entities are needed to be involved throughout the whole computation process. Let us illustrate this by an example: suppose we want to perform a task, and in order for the task to be completed, it requires two pieces of data, D_A and D_B, which are stored on two different sites S_A and S_B respectively. In the client-server model, both D_A and D_B needs to be sent to the server nodes and compute, whereas if the mobile agent technologies are in use, D_A can be sent to S_B directly and complete the task at S_B (or vice versa) without the participation of the intermediate server.

This peer-to-peer approach has a major benefit: a computation without any participation of third party entities is generally believed to be more secure. Let look again at the above example: suppose that S_A trusts S_B for the computation and the data it provides, but not the server. In this scenario, for the approach of using client-server model, we must establish the trust relationship between the server and S_A/S_B before the computation can be done. However, for the case of using the mobile agents, this *redundant* trust relationship is not necessarily be established: the two involving entities can share the data and perform the computation themselves. And even if S_A does not trust S_B for computation (but still trust it for data), S_A can still retrieve the data from S_B and do the computation locally. In other words, in the scenario where mobile codes are present, trust relationships are not forced to be established just because of computation issues. The location of computation can be made dynamic through using the peer-to-peer computing feature.

Above is just one of the advantages in using the peer-to-peer approach. For more discussion on the issues regarding peer-to-peer computation and its relationship with mobile agents, please refer to Section 2.4.

2.1.5 Service customization

One question may immediately arise from the previous scenario. If S_B can provide the computation service that S_A wants, since S_A trusts S_B on performing computation, the above problem on trust relationships can then be solved easily be sending D_A to S_B using just the traditional client-server approach. It is true. However, how can one make sure that the trusted site (i.e. S_B) *must* provide the service that one requires? It is impossible to do so in the client-server approach. In the client-server model, server plays an active role while client takes a passive role. In other words, in such a scenario, servers will provide an *a-priori* fixed set of services, while client can only choose among those service provided and use. In that case, it gradually evolves to a scenario that the service provider at the server-side must listen to what the clients want in order to update the server and provide the corresponding services, and the client never satisfies with the services provided because that is not exactly the service he or she wants – one service can never satisfy all users' needs. Also, an immediate side effect of this is that the complexity and the size of the system will be increased, directly decreasing the maintainability of the overall system.

This problem can be solved if the end-users can customize what they want to do on the server. This is where *service customization* comes in. As we have just said, service customization is not possible to be supported in the client-server model, but in contrast, it is a natural application and benefit in the era of code mobility. With code mobility, users are now able to write their own service, or to compose the services from code components themselves, making them able to obtain a customized functionality that meets their specific needs. The code for this self-composed application/task can then be sent to the server-side and perform what the end-users exactly want. By using this approach, the resulting system can be made much more flexible in terms of functionality.

2.1.6 Deal with data with different mobility natures

With the introduction of code mobility, many computation scenarios that can not be nicely addressed by the client-server approach can now be solved more naturally and comprehensively. Among them, many problems arise from the *data mobility* issues. In the pervasive computing world, there are in general two types of data, each categorized by their mobility in the environment: *transferable* data and *non-transferable* data. Transferable data refers to the data that are freely movable over the network, whereas non-transferable data are those that cannot be moved out from its hosting site. Typical non-transferable data include some private data, and large-sized data that is too time-consuming, if not impossible to be transferred out of the bandwidth-limited environment. Typical client-server interaction approach clearly cannot address the issues of non-transferable data in simple ways, since it must find a way to move the *non-transferable* data to the server for computation. For example, in order to deal with data privacy, a trust relationship must be established between the client and server beforehand; and in order to move large-sized data, some special techniques like streaming or compression may need to be used. In either case, the complexity of the server design would be greatly increased (with the problem not fully solved, e.g. the client refused to establish a trust relationship with the server, or it does not have enough memory resources to perform the corresponding data streaming or compression strategy).

The introduction of mobile code can solve the above problem in a nice way. For the case where the non-transferable data is located at a remote site (with no computation codes there), an mobile agent or simply a mobile code can be sent to that site and perform local interaction and computation with the involving data. After that, the computed data that the agent owner needs can be brought back. In most cases, the size of the computed data being brought back is smaller, thereby solving the large-sized data problem. Similarly, for the case where the non-transferable data is located at the local site, one can bring in the required mobile codes from other remote sites and perform the execution.

To summarize, with the introduction of code mobility, code, instead of data, becomes the subject of movement, and can move to the destination site to perform local interaction with the data. Since the data involved do not need to move, it solves most of the data mobility problems naturally.

2.1.7 Execution delegation

Mobile devices always have constraints on resource usage. These constraints are mainly placed by its small form-factor and battery power consumption. In other words, it will be power-consuming to run an application that needs long or intensive computing power. In that sense, execution delegation is a valuable feature in addressing the resource-shortage problem on running these long-lived applications in the pervasive computing environment. By execution delegation, whenever there is not enough resources currently available on the local device to complete computing a task, the whole or part of the computation can be moved to a remote server that has the required resources to perform the computation. By doing so, not only the resources usage of the client devices are reduced, the execution time of the code can also be shortened.

2.1.8 Automatic updates and maintenance on functionality

As mentioned above, in conventional distributed systems built following a client-server model, the servers provide an *a-priori* fixed set of services accessible through a statically defined interface [6]. It is often the case that this kind of static functionality is not suitable for unforeseeable client needs and for some further extensions of the system. For example, in the pervasive computing world, algorithms and protocols are rapidly evolving in terms of speed and the range of functionality they provide. It may be the case that the algorithms and protocols now considered to be fast and robust become slow and obsolete tomorrow. Therefore, a typical way to address this problem in the client-server approach is to upgrade the server with the newest functionality. This approach, however, requires human intervention on the local site. Furthermore, using this scheme will only increase both the system's complexity and size, without increasing its computational flexibility.

With the introduction of code mobility, the functionality of an application (no matter it is a server application or a client one) can be upgraded in a simpler manner. The service provider / application design can send out an mobile agent carrying all the routines that are necessary to carry out the required updates. When the mobile agents roams across a set of hosts, the corresponding updates on the applications can be performed according to the capability of the target client device and its surrounding environment.

On top of using the *push* approach to perform the necessary updates, a *pull* approach can also be used. Suppose that code components can be dynamically linked and formed an application (similar to our *facet* abstraction). In that case, these code components can be downloaded on demand from the code repository in order to compose the desired application. There are two main benefits on using this approach. First, the updates on the corresponding code components can be made in their own code repository. It can reduce the efforts being used on updating different parts of an application. Second, since the required code components are brought in on demand, no extra efforts and resources would be wasted on updating irrelevant parts of code. In that sense, the most up-to-date version of the code component would be loaded upon request, and the changes performed by the update are virtually done in a lazy manner. This way, the updating of the application can be done automatically.

2.1.9 Remarks

Of course, the points stated above is just an non-exhaustive list of advantages of LMA in some typical scenarios in the pervasive computing environment. Other common advantages of the mobile agents, such as *reducing communication traffic, overcoming network latency* through local interaction, *balancing the network load* and *encapsulating protocols,* etc. are not mentioned here. As one can foresee, with its flexibility, the different patterns of use on mobile code technologies can really bring a wide range of new possibilities to the application designers for pervasive computing.

As a point of note, in this section, we are *not* trying to convince the readers that the computing models being introduced by the mobile codes are all superior to those brought by the traditional client-server approach: there are certain scenarios in which adopting the client-server approach is much more beneficial than using mobile codes, e.g. the client-server approach is usually considered to be more efficient than the mobile code approach, since the static code can be optimized towards the server architecture and its environment, and there is generally no need to wait for the code loading time. Instead, we just give the readers another view what computing models can be brought and how flexible it can be by introducing mobile codes into the picture that is *complementary*, but not *contradictory*, to the existing client-server approach.

2.2 Brief analysis on several mobile agent systems

We have seen from the previous sections about the benefits of using MASs and MCSs. In this section, we would look into some practical implementations of the MASs, and try to analyse their characteristics, as well as their strengths and weaknesses. Also, we will try to compare our mobile code system with them in terms of features supported and implementation issues, at the same time identifying the work needed to be done in our mobile code system.

2.2.1 MASs in real world

There are currently many MAS implementations available in the world (One can find an approximate list of MASs here [11]). Since mobile agent has to address the issue of platform heterogeneity, they are usually implemented using platform-independent programming languages. We can mainly classify them into two types. One type is implemented based on some interpreted research languages like Tcl, Scheme, Oblique and Rosette [13]. Telescript by General Magic, the first commercial mobile agent platform as it claims to be, Agent TCL, TACOMA, ARA are examples of this type. Since the year that Java was launched, dozens of Java based MAS have been developed, which formed the other type of MAS category. In fact, the Java Virtual Machines and the class-loading feature, coupled with techniques like object serialization, RMI, multithreading and reflection, have made the building of MASs much simpler. In this section, we are going to analyze three Java-based MASs, with one of them is a commercial implementation and the other two are research-based.

2.2.2 IBM Aglets

Aglet is a commercial MAS developed by Tokyo Research Laboratory of IBM, yet it is open-sourced recently. It is one of the most widely used MAS in developing mobile agent applications, and has definitely received the most press coverage [13]. It is named as a combination in meaning

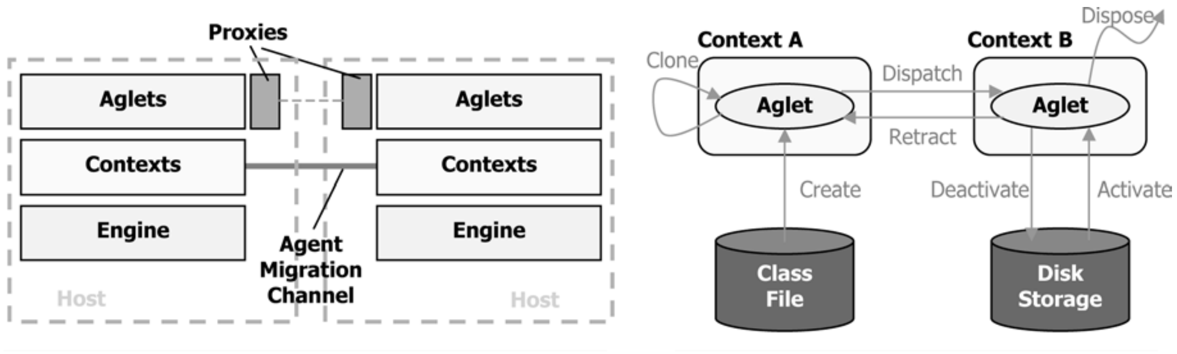

Figure 2-1: The Aglet architecture and its interaction model

of "Agent" and "Applet". It uses typical Java features such as object serialization and RMI, and provides a graphical Aglet control interface called Tahiti [15].

The Aglet runtime architecture is a three level hierarchical structure: *Aglet*, *Context*, and *Engine*. *Aglet* is the mobile agent object to be used in the MAS; *Context* is a static execution environment for multiple aglets; *Engine* is the JVM running on a host which supports one or more Contexts. For reasons of security, limited access and location transparency, there is an addition structure in Aglet architecture called *Proxy*, which serves as a representative of Aglet and interact with the outside world. The overall Aglet architecture is presented in figure 2-1.

With this execution environment, six critical operations can be applied to aglets during its lifetime: *creation*, *cloning*, *dispatching*, *retraction*, *deactivation/activation* and *disposal*. ***Retraction*** here refers to pulling the aglet from its current context back to the requesting context, and ***deactivation/activation*** are used to temporarily halt/restart the execution of an aglet by storing the associate state in the hard drive (using Java object serialization techniques). The relationship between these operations is also presented in figure 2-1.

Basically, Aglet adopts the *event-driven programming model*. Aglets would have some associated listeners that catch events in the life cycles of an aglet and process accordingly upon reception of these events. Typical events include *clone event*, *mobility events* and *persistent events*. Also, it supports messaging mechanisms like synchronous messaging, one-way messaging (messaging without ack) and future messaging (asynchronous response) [15].

As one can see, Aglet deploys most of the functions of typical mobile agent applications. Compared with other MASs, Aglet has its own selling points: its package is easy to install and it has an easy-to-use GUI for controlling agents. These make the world of agents easily accessible to even Java novice. However, as it is an old implementation of the mobile agent technologies, it is rather inefficient (a common weakness for mobile agent systems). Also, 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 codes not maintainable.

Our mobile code system also have a similar architecture as that of Aglet: with JVM at the bottom, then LMCS acting as the context, and then LMA playing the same role as aglet. However, different from the Aglet's architecture where a proxy object is needed for agent communication, by the pluggable feature of the LMA, we can plug in any different functional components inside the LMA on demand to have different communication protocols, which makes the resulting communication scheme between aglets more flexible than that of aglets. On the other hand, contrasting the Aglet approach, since the aim of implementing our MCS is not on deploying a full-fledged MASs, our MCS does not have any GUI support for controlling the agents (we plan to add that in the future). However, our mobile code system support strong mobility, so that the agent can migrate at any points during its execution. This makes the power of the mobile agents be fully deployed. And because of the strong mobility support, execution can be suspended and migrated at any point, there is no need to use the event-driven model, which predefines and limits the execution entry point after agent migration, retraction and disposal. Finally, since Aglet does not plan to work in the pervasive computing environment, it seems not have any strategies on making the mobile agents lightweight, which deviates from the design principle of our mobile code system.

As a remark, most commercial implementations of MASs (e.g. ObjectSpace's Voyagers [31] and IKV++'s Grasshopper2 [3]) exhibit only weak mobility features, and they all have a event-driven programming interface similar to that in Aglet (in fact, the programming interfaces in these languages more or less stick to a standard, called MASIF, which will be discussed in Section 2.3.2).

2.2.3 JAMES

James MAS [29] 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, where performance is an important issue. Since performance optimization is one of its main goals (and also one of the requirements of MAS), James was implemented using a lot of performance optimization techniques, including *code caching and prefetching* schemes, and *thread and socket pooling* [30]. They are described as follows:

- **Code Caching Schemes.** Every nodes needs to fetch the agent code from the code server in order to execute agents. However, the central code server approach being used by many other MASs had proved to limit the performance of the MAS. A hierarchical code-caching scheme is therefore proposed in James, aiming to offload the central server by decentralization and thus improving the performance. The caching scheme is as follows: First the agent classes are searched in the main memory cache, to see if the bytecode of the class has been preloaded; if miss occurs, the platform would search the local disk (cache) to see if the JAR files for the agent classes are downloaded; if miss still occurs, the local system would try to get it from the previous system on the itinerary; if miss again occurs, the local system would contact the main code server. Some replicated code servers may also be used to increase the availability and increase the scalability of the system.

- **Prefetching Mechanisms.** When an agent is created and launched to the James platform, it has its own itinerary of visiting sites. By analyzing this, a James Manager could help to prefetch and preload the code to relative sites, or even preopen the migration channel by sending lightweight messages, so as to decrease code loading time.

- **Thread and Migration Channels Pools.** It was found that object creation and connection establishment take up substantial amount of execution time. Through reusing the existing resources from resource pools, performance can be improved.

Although by using these strategies, James can achieve better performance than the traditional mobile agent systems, the techniques it uses are mainly suitable only for telecommunication purpose, but cannot be applied for general mobile agent purpose. It is mainly because in real situation, there cannot be a central manager for predicting the agent itinerary. This centralized approach would quickly become the performance bottleneck and affect scalability. Furthermore, centralized control would decrease the freedom of the agent and thus the overall flexibility of the system.

On the other hand, as the intelligence of the mobile agent grows, the itinerary of the agent will dynamically depend on the surrounding environments. It may not be possible for the central manager to predict the next moves of the agents, thereby making the prefetching mechanism fail to improve the performance.

In view of these, in our architecture, no performance optimization was done on the MASs of the agent sites, since we believe that the agent itinerary is generally unpredictable. Instead, the optimization is done at the code repositories distributed over the network (which eliminates the problem of centralized control). they will predict what code components the agent will use in the future and prefetch it before hand, thus making code fetching procedure more efficient. In fact, these code repositories are the Intelligent Proxy Servers in our architecture.

In fact, some schemes like using compressed JAR for the migration code of the mobile agent (which trades *computation time* with *code transfer* time) were undergoing research in the James platform. It was also suggested (but not implemented) that a special type of agent called *elastic agents* could be used: they can drop some code when they do not need it, or load some new code when they need. This approach is in line with how our LMA is designed and implemented under the same goal on saving the communication bandwidth. But our scheme push one step further, instead of using the code components, we use our facets instead, which themselves can be adaptive to the surrounding environment. The overall network bandwidth can therefore be further reduced.

2.2.4 NOMADS

As mentioned in Chapter 1, Java based MASs would exhibit the problem of lacking execution state capturing feature. Owing to this, for MASs that want to provide strong mobility and resource tracking feature, they have to either modify an existing Java VM (adopted in Sumatra and Ara MASs [22], which has an disadvantage of limiting the ability to redistribute the system due to licensing constraints on JVM) or using a preprocessor or compiler to add state-saving code inside it (adopted in WASP MAS [32], which does not work well with multiple threads). These are the problems where the design of the Nomads MAS comes in to address.

The Nomads MAS [35][36] is composed of two parts: an agent execution environment called *Oasis*, and a custom VM called *Aroma*. Through using the Aroma VM, which supports execution state capture and resource control, two key enhancements over other Java agent environment are enabled: *strong mobility* (which supports forced migration) and *safe agent execution* (the ability to control resources consumed by agents, facilitating guarantees of QoS while protecting it from Denial of Service attack). However, there were still limitations in the Aroma VM: because the Aroma VM adopted the approach of mapping Java threads to native threads for execution (based on some design decisions), the state-capturing mechanism became more complex (i.e. have to maintain their mapping correctly), inefficient, and less precise and well controlled (because the state capture feature of native threads is not available, so if any of the Java threads are executing the native codes, the Aroma VM has to wait for the threads to finish their native code before initiating the state capture).

There are still some inadequacies in the Oasis platform. Current Oasis would run each agent within a separate instance of Aroma VM for easy support of resource accounting. However, such approach would increase system resource usage. Also, the mechanism of dynamic adjustment of agent resource limits has a problem in reclaiming the resource that is currently being used by that agent. As one may expect, because of the additional execution state information of the agent that is being captured and transferred, the performance of Nomads MAS is 1.87 to 3.7 times worse than other MASs [36].

Instead of using the VM modification method to implement strong mobility, our mobile code system uses the source code instrumentation method by using a preprocessor or compiler to add state-saving code inside the agent code, which is mentioned above. The main benefit of this approach is that the resulting agent programs can be readily executed on heterogeneous platforms over the standard JVM, which is different from the VM modification scheme in making the Java-based MAS no longer portable. The problem mentioned above: the issue of migrating of multi-threaded states has already been mostly addressed by our strategy discussed in this Thesis, although at a greater expense than the VM modification approach in terms of memory usage and resulting performance.

2.2.5 Remarks

As a remark, since the MAS technology is still under steady growth, the metrics on measuring how good a MAS is are still unknown. At present, what we can conclude from looking at all the above MASs is that, the above MASs have their own standing points for them to become competitive: Aglet's ease of use, James's efficiency, and Nomads's support of strong migration and dynamic resource management. Because of their different goals, these systems have taken into account a lot of tradeoffs between functionalities and efficiencies in their design. Therefore, until standard metrics specifying what features should be included in a good MAS are identified, it is hard to say and unfair to compare among different MASs, since they have rather different goals and ways to accomplish it. Nevertheless, we believe that our mobile code system has already been designed to deal with some of the most important issues in the MAS researches: the size reduction in agents and the *portable* support of strong mobility on multi-threaded agents. In the next section, we are going to explore the possible fields that has been opened up by the introduction of mobile agents and mobile codes.

2.3 Areas of researches brought by mobile codes and agents

Although it has been known for years that the use of mobile agents and mobile codes has many benefits to the global network community, and there are already many MASs developed for it, due to the immaturity of the current mobile-agent technology, there are some potential limitations in porting it to WAN or even the Internet. In recent years, active researches have been done on various related areas in order to make the true power of the mobile agents technologies be exploited. Besides the area concerning agent mobility (which takes care of the issues on supporting different kinds of agent mobility, how to make the agent more readily transferable over the network, etc.), there are four other main areas that current researches focuses on. We will briefly discuss them in the following subsections.

2.3.1 On knowledge representation

There is a problem in implementing how agents represent their knowledge and communicate their needs with other agents in order for them to *meet* and *collaborate* [13]: the static interface of current mechanisms for mobile agents to meet and collaborate seems to be quite limited in the area of flexibility and possibilities for agent interaction. Some comparatively promising work in this domain is the Knowledge Query and Manipulation Language (KQML), and its related efforts, languages, and systems (KIF, Ontolingua, and others) from the Knowledge Sharing Effort (KSE) Consortium. ***KQML*** is a both messaging format and handling protocol for supporting runtime knowledge sharing between agents. This is a kind of language that is commonly termed as *Agent Communication Language* (ACL). It helps the agent system to get to the complexity level of communication that will be demanded by applications like E-Commerce. However, the drawback of this complexity is the difficulty in integrating it to the agent systems. *Ontology* is yet another important aspect in agent communication technology. It is a *specification of conceptualization*. Different from XML that makes data only meaningful to person by means of tagging, ontologies make data meaningful to *both person and computer* by means of semantic relationships. An example for this is to use DAML (DARPA Agent Markup Language) with OIL (Ontology Interchange Language), which extends the Resource Description Framework (RDF). However, this technology is still in its submission stage. Overall, the immaturity of it limits the growth of the mobile agent community.

2.3.2 Portability and Standardization

Addressing platform heterogeneity is one of the requirements of the MASs, thereby retarding the progress of interoperability and rapid deployment of the mobile agent technologies. The movement of agent onto different computing platform is not as easy as it may seem. One common approach of implementation is to compile them into some intermediate platform independent representations (such as bytecodes in Java), and then later either just-in-time compile them into native code, or execute them inside an interpreter. However, for mobile agents to be widely adopted, the intermediate code must be portable across different MASs, and therefore significant amount of standardization in the format of the mobile agent has to be done.

There are mainly two agent standardizations efforts in the current mobile agent literature. One of them was the Object Management Group (OMG)'s Mobile Agent System Interoperability Facility (MASIF), which mainly standardize four areas of technologies: agent management, agent transfer, agent and agent system names, and agent system types and location syntax [15]. The intention of this standard is to achieve a certain degree of interoperability between mobile agent platforms of different manufacturers. A MASIF-compliant agent platform can be accessed through the standardized interfaces that are specified by means of the OMG's Interface Definition Language (IDL).

Besides OMG, the Foundation for Intelligent Physical Agents (FIPA) is another promising standardization body in the context of agent technology. While MASIF focuses on standardization on issues related to the mobility of agents, the standardization efforts of FIPA were more associated with the intelligence of agents. In this context, one essential achievement of FIPA is the specification of a sophisticated Agent Communication Language (ACL). It standardizes issues on agent management, agent communication language, agent/software integration, etc. By doing so, it is now possible that the construction and management of an agent system composed of different agents built by different developers. Agents can also communicate and interact with each other to achieve individual or common goals. Even more, it allows the legacy software or new non-agent-based software systems to be used by agents.

The main differences between the two standardization specifications are essentially that: MASIF is based on agent platforms and it enables agents to migrate from one platform to another, while FIPA spec is based on remote communication services. While the former is primarily based on mobile agents traveling between agent systems via CORBA interfaces and does not address inter-agent communication, the latter focus on intelligent agent communication via content languages and do not say much about mobility. To be specific, FIPA adopts an agent communication paradigm, which can better express the nature of cooperation and is more suitable for integration with other AI technologies. On the other hand, MASIF adopts a mobile agent paradigm which is more appropriate in situations where dynamic and autonomous swapping, replacement, modification, and updating of application components are required. Despite this, both FIPA and MASIF provide their specifications in an implementation-independent way, so as to allow the agents and its execution environment be implemented in different ways, while retaining their ability to communicate with one another.

Although these two standard specifications address different issues related to mobile agent technologies, a new emerging standard, FIPA2000, is already proposed to integrate the two. One benefit of this integration could be ACL support in MASIF-compliant mobile agent environments. On the other hand, the FIPA standards could benefit from the mobility aspects handled by MASIF. In either case, the mobile agent communities and its related technologies must advance and become more mature. As a reminder, the Grasshopper-2 MAS [12] has been available since 2000 and is both MASIF- and FIPA-compliant through using MASIF and FIPA addons (although it is still a weak mobility MAS).

2.3.3 On agent security issues

Agent security has long been an annoying problem to the mobile agent technologies, and it is the main barrier from letting mobile agents be widely deployed and accepted, although it is undoubtedly an important issue for MASs to work in an open environment such as the Internet. Traditional security mechanisms dealing with non-mobile codes may not be applied in this situation. It is because the non-mobile codes are always executed on a *trusted* local environment. Since agents in open-area-MASs may belong to different domains, the security problems thus arise. There are in general three mutual security problems with respect to mobile agent technologies: (1) Agents must be protected from possible attacks during transit; (2) Hosts must be protected from malicious agents, and (3) Agents must be protected from malicious hosts. Similar to traditional security studies, the agent security should address different issues on *confidentiality*, *agent integrity*, *mutual authentication*, *authorization and access control*, and *auditing and resource accounting*. Among the three scenarios mentioned above, (1) is the easiest one to be addressed, where most of the security issues can be solved by some simple use of cryptographic technologies. However, for the scenarios in (2) and (3), since codes has to be executed on the host, if the relevant security measures cannot guarantee that the behaviour of the execution is the same as expected, many potential attacks can be made possible. We will look into these two scenarios in the following paragraphs.

Protecting the host from malicious agents

The problem on *protecting the host from malicious agents* has received considerable attentions since the day it was born. The main issue here is about the authorization and access control on the agent execution. If this issue is not tackled properly, it simply creates equivalent, or even more disastrous, damages to the agent-hosting hosts when compared with those created by the viruses: virus unfortunately is a specialization of mobile code[1].

There are currently two complementary solutions to this problem. One is to exercise *resource accounting and control* on the programs executing on the host, e.g. limiting the disk quota, CPU cycles or number of network connections to be made by the hosting agent code. This can be done by loading in some resource constraints specifications when the MAS starts (e.g. the `java.policy` files used in Java). Doing so can also have a positive side effect on preventing agents from incurring large bills to agent owners in the E-commerce application, if a billing model exists and counts the resource usage of an application.

Another is to apply some sort of *code signing* technologies. Theoretically, this can be done by letting a trusted third party to test the correctness of the code and digitally sign on it. However, it is generally difficult to find a third party that can be trusted by all the participating agent-hosting hosts. Even worse, it is also difficult to know ahead of time which hosts in the world are the participants in the agent execution, since the mobile agent may determine its itinerary dynamically. In view of this, current mobile agent technologies address this issue through the use of *Proof carrying code* (PCC) [21]. In a typical instance of PCC, the agent host will establish a set of safety rules that guarantee safe behavior of programs, and the agent programmer will create a formal *safety proof* that proves, for the untrusted code, adherence to the safety rules. Then, the host is able to use a simple and fast *proof validator* to check, with certainty, that the proof is valid and hence the untrusted code is safe to execute. In this case, the agent is said to be *self-certified*, and does not need the involvement of trusted third parties.

[1]Even worse, some viruses have fixed *signatures* for anti-virus software to detect, while mobile codes, in general, have not. In view of this, mobile codes/agents are believed to be able to create much more damages if their access rights are not handled properly.

Protecting the agent from malicious hosts

The last problem, *to protect the agent from the malicious hosts*, is the most difficult problem among the three. It is because the agents are executing on a host, and the host has to have access to all the state and code of the agent in order to perform execution. Therefore, it is technically very difficult to hide anything about the execution from the host [4]. Based on this, many possible attacks can be done. For example, let us consider the case when a mobile agent is deployed to shop for a certain item on the Internet. If the mobile agent is navigating on a malicious host, the host may try to read and modify the offers from the other hosts (which are carried in the data states of the mobile agents) or even change the execution flow of the agent code in order to force it to take the offers from the malicious host. The consequence of this can again be disastrous.

Besides the pessimistic approach that always migrates the agents to trusted hosts before execution, there are two common approaches proposed to address this problem. The first one involves the use of the so-called *mobile cryptography* [25][23]. In this approach, besides the data carried by the agents are encrypted, the *agent programs* are also “encrypted” in such a way that it consists of program statements (or operations) that can work on encrypted data, so that the output encrypted data, when decrypted, would be the same as that generated from normal unencrypted computation. Any tampering or modification to the computation process would result in invalid answer (or response that cannot be decrypted). However, such an approach exhibits certain limitations: it cannot be applied to random programs, currently only *polynomial* and *rational* functions can be used for this purpose. Also, since this scheme depends on data encryption, if there is a way to decrypt the output of a protected agent, it is likely that the agent can be attacked too (Notice that authorizing one to view the data is not equivalent to trusting the same entity to do the computation: they should be specified separately in the security policy).

Another approach is to use the so-called *mess-up* or *obfuscating* algorithms. These algorithms converts a program and its data part of an agent into another agent that does basically the same as the original agent. However, the resulting agent code is difficult to analyse in terms of programs that examine a program like *flow analysis* or *program slicing techniques*. However, it does not have any formal proofs on the strength of protection of such an approach in order for this method to be fully deployed in the environment.

In conclusion, up to now, there is no way to fully address this issue on protecting agents from malicious hosts. However, again, in order for mobile agent to be widely deployed and flexibly used, this problem needs to be solved. At this moment, we take the pessimistic and less flexible approach, which finds a trusted site before execution.

2.3.4 On agent controlling strategies

Yet another area of research is on the strategies in controlling the mobile agents. By controlling the agents, we refer to the mechanisms that are: to *locate an agent* so as to get information about the current status of it; to *terminate an agent* that is no longer use; and to *detect the orphan agent* and take appropriate actions when the agent is no longer reachable from its initiator. This is an area that is rarely discussed. In most of the current implementations of MASs, only simple strategies like brute force searching or randomly guessing techniques are used to locate the agent, which clearly cannot be tolerable in the open environment like the Internet. There are in general three strategies used in controlling the agents. All these are presented in [2].

The first strategy is the *energy concept*. In this concept, every agent initially has a certain amount of energy. Every action or every service it requests takes it energy to complete. When an agent has no energy, it is classified as an orphan and can be terminated. Of course, before the agent is terminated, the agent can refill its energy by sending request to its initiating host. Obviously, this strategy can only address the problem of orphan detection.

Another strategy that is commonly used is the *path concept*. It is in fact similar to the mechanism in tracking mobile objects in distributed systems. Every mobile agents will leave a *footprint* on every hosts along its itinerary when they migrate from hosts to hosts. The resulting path can then be followed to find the agent and/or to terminate it. In contrast to the energy concept, the path concept addresses the issues on locating the agent and terminating the agent, but is unable to detect the orphan agents.

In order to control the agents, the Voyager MAS [31] adopts the hybrid approach between the two strategies. In Voyager, it uses the concept of *virtual objects*, which essentially plays the same role as *footprint* described above. When messages (e.g. for locating the agents) are to be sent to the remote agents, they are first sent to the virtual object. The virtual object would forward the message to the remote side. The opposite occurs when the message needs to return an acknowledgment. Under this model, when the mobile agent wants to move, it would wait until all pending messages are processed, and then move to the specified destination, leaving behind a virtual object for forwarding messages. Of course, after the agent migrates, the path left behind can be optimized (shortened) by removing the redundant subpath after ensuring that no messages for that agents are on the way sending on the subpath.

On the other hand, Voyager also support orphan detection to claim back system resources. It uses a variant of the energy concept: instead of requesting the initiating host to refill its energy, the virtual object will take the responsibility of the initiating host to refill its energy. The virtual objects are specially programmed such that they would send a *lightweight heartbeat* (i.e. energy re-filler) to the mobile agents in regular intervals. Virtual objects would be garbage collected in normal manner, and if no heartbeats were sent to the real object within a certain interval, the mobile agent would be considered as orphan and would kill itself. This approach is better than the energy approach in that no permanent connection between the agents and the initiating hosts is needed to be maintained. The initiating hosts can run only intermittently to check for its status, and go offline as it want before the virtual object is garbage collected.

2.3.5 Remarks

All the technologies mentioned in this section are still under active research, and are rapidly and continuously evolving in the mobile agent technologies. Despite this, even if these technologies are evolving, because of the *pluggable features* of our LMA, any new strategies should be readily incorporated into our system – which is also one of the benefits in using our design of LMA. In the future, we may put our concentration on designing good pluggable interfaces for all the above operations in order to exploit the benefit of our LMA (Currently we have only defined simple plug-in interfaces for the above operations).

2.4 Peer-to-peer computing and mobile agent

As what we have discussed earlier, peer-to-peer computing is drawing increasing attention in the area of computer science. Although up to now there is no common agreement on the definition of the term *peer-to-peer computing*, it is generally believed that peer-to-peer systems enable two kinds of user actions: they all enable their participating peer nodes to *share their resources* and *collaborate* to get their own goals achieved. On the other hand, in terms of architectural designs, they also exhibit the following common features:

- Systems with peer-to-peer computing feature are always characterized as decentralized, self-organizing distributed systems because of the absence of a centralized server involving in the communications between two parties. Also, in usual cases, peer-to-peer applications have their own naming scheme that is independent of the DNS naming scheme, which relies on the presence of centralized DNS servers.

- In the peer-to-peer computing world, each participating entity has the *same accessing capabilities*, and each of them has the *same right* in initiating a communication session with any other entities. In other words, they are *symmetric* in terms of functional capability. Variable connectivity between peer nodes can therefore be allowed in the peer-to-peer systems.

- The peer-to-peer systems are designed in such a way that resource sharing and accessing can be done in an *automatic* yet *anonymous* manner. This guarantees that resource accesses can be done in a *private* and *unbiased* way.

All the features mentioned above, including decentralized architecture, anonymous and automatic sharing and collaboration, symmetric accessing capability, etc., perfectly suits the needs of users in the pervasive computing environment. In pervasive computing environment, users can enjoy accessing their required resources anytime, anywhere on any devices. If peer-to-peer systems can also be designed for mobile devices, through using the peer-to-peer services, these characteristics of pervasive computing can be enabled in no time. In view of this, enabling peer-to-peer feature on mobile devices seems to be an indispensable feature in enabling pervasive computing.

Conventional way in enabling the peer-to-peer feature involves the use of a *broadcast-based* communication protocol: whenever a user wants to get some resource, they just need to send out a resource-request message to its peer node. If the peer node has the resource, it would send an acknowledgment back to the originating node. But no matter the node has the requested resource or not, it would again forward the message to its peer node. The whole process is repeated until a maximum searching depth is reached. The user can then select one of the nodes in fetching the required resource. Of course, in order to ensure the anonymity of the system, it is not a direct broadcast approach: some mechanisms (not detailed here) has been added into the communication strategy. Nevertheless, this approach in general still consumes a lot of communication bandwidth.

With the use of mobile agents, the bandwidth usage of implementing a peer-to-peer scheme is expected to decrease. It is mainly because instead of using a passive entity like the request message in the conventional approach, the active entity: *mobile agent* is being used in sharing and collaborating between the two communication ends. This makes the principle of underlying message delivery mechanism different. While some message analyzing and forwarding services must be present in the message-based peer-to-peer systems, no such services are needed in the agent-based peer-to-peer systems. By using the same principle as *service customization*, the mobile agent would *actively* find out the resources it needs through executing its code on the target system. With the information provided by its owner describing about the resource he wants to find, and if possible, which domain to find, the intelligence of mobile agent can help its owner to negotiate and find resources that is best suitable for its owners' need, at the same time trying it best to find an itinerary that is most probable to get what user's need. By its intelligence, the agent needs not travel over all the peer nodes, and therefore the overall bandwidth usage of the network can be reduced when compared with the broadcast-based approach.

On the other hand, in the traditional message-based peer-to-peer systems, the location of the message originator must be specified in some form. But in order to ensure the anonymity of the system, this information has to be hidden using some special mechanisms. This scenario will not happen in the mobile agent system. By making the mobile agent anonymous (i.e. hiding the identity of the mobile agent) and simply using the agent tracking strategy built in, the required resource can be retracted together with the agent without violating the anonymity of the system. In fact, besides bandwidth saving and anonymity enabled, some other benefits in using mobile agents, e.g. disconnection tolerance and automatic behaviour, are also advantageous in the peer-to-peer environment. If the adaptiveness (in terms of both resource usage and functionality) of our LMA can be incorporated in the future mobile agent systems, and that the limitations of the mobile agent stated in the previous section are solved, the use of mobile agents is potentially a perfect candidate in supporting the peer-to-peer technologies.

2.5 Grid computing and mobile agents

Managing access to computing and data resources is a complex, time consuming and tedious task. In real life situation, no users or researchers want to spend lots of time on deciding which systems to use, where the data resides for a particular application domain, how data migrates, and the required data rates in maintaining the correct functioning of the application. They want a system to get these all *automatically* done for them, with optimized performance and resource usage. This is where the concept of *Grid computing* comes in.

The *Grid* is proposed as a means to seamlessly integrate computing systems and clusters, data storage, specialized networks and sophisticated analysis and visualization software. Like an electrical power grid, the Grid will aim to provide a steady, reliable source of computing power. In the same way that we can utilize electric power without actually owning power generating stations, the Grid will offer us the ability to connect to high end computational engines or clusters without owning the engines for computing (*Distributed supercomputing* and *high-throughput computing*), or the data repositories on which the computation is to be performed, or into which results are to be stored (*Data-intensive computing*).

No matter which services the grid will offer, it is by no doubt that the data and computation migration will play an important role in the Grid computing world. Middleware that automatically allocates the resources on demand and based on user-preferences are believed to be the enabler of the Grid computing technology. In that sense, the mobile agent/code technology is again one of the best candidates in serving this purpose. Through the peer-to-peer support of the mobile agent technologies, the grid is now not only a low level infrastructure for supporting computation, but can also facilitate and enable information and knowledge sharing at higher semantic levels. Besides, features of the mobile agent technology that are also beneficial to the peer-to-peer technologies such as platform independent execution, lower bandwidth consumption, disconnection tolerance, load-balancing and flexible computation models make it an attractive alternative for implementing a heterogeneous grid computing system.

Although mobile agents have a lot of potential benefits in use with the Grid computing technologies, in our opinion, there are still two major shortcomings in the mobile agent approach that must be overcome before it can be deployed in the Grid computing world:

- **Security.** In the current implementations of Grid computing technologies, mobile codes are rarely used. Therefore, traditional static security protection strategies such as the public key infrastructure can be readily deployed in this environment. The only problem that needs to be addressed is the design of inter-domain security policies, which are to solve the problems like *N-way security contexts* and *local heterogeneity* [5]. However, when the presence of mobile code is taken into account, even the most fundamental security problem related to mobile agent technologies (how to protect mobile agents from malicious hosts) cannot be fully solved, as we have discussed in Section 2.3.3, not to say addressing the issues on cross-platform security policies. The supporting security scheme is still too premature to be used in the Grid computing world.

- **Performance.** Grid computing, with *distributed supercomputing* as one of its common goals, generally requires its implementing systems to have relatively high efficiency in using com-

> puting cycles. However, although the use of mobile agents is good for cycle and resource sharing (and provides *functional flexibility*), the performance of the MASs may be rather slow, where computing cycles are generally wasted in loading codes and migrating agents. This is not a problem in the area where asynchronous interactions are useful, but in order for the mobile agent technology to be fully deployed in the Grid computing environment, this may become a barrier that must be overcome.

Despite these technological barriers, one can foresee that the philosophy behind adopting mobile agents in global (pervasive) environment and the goal of the grid computing in general coincides. The following paragraph is quoted from an article [28] which expresses the author's view on the future of using Grid related technologies.

> All computing resources will be made available and configured dynamically for a specific task at hand, or for a virtual organization's specific requirements. Applications will be put together to match the specific functionality required. Data will be accessed wherever it resides. Processing power will be delivered on demand. And the users will be insulated from the complexity of the underlying systems, enabling them to easily access the resources they need to get their jobs done.

This also stay in line with the goal on designing our Sparkle architecture: the small functionality components can be put together to form a new and customized functionality, which can be plugged into and carried by the LMA to use the corresponding resources and processing powers of the nodes over the pervasive network. Also, the internal complexity of driving mobile agents is also hidden from the end-users, making them easier to get their jobs done. Again, if the security and performance of the mobile agent technologies can be improved, the use of them can be very much beneficial to the Grid computing technologies.

2.6 Other related projects

There are in fact some other projects that aims at similar goals as those in our design of the mobile code system, especially on the issues of addressing the user mobility in the pervasive computing environment. However, most of them do not involve the use of mobile codes or agents, and do not even need to migrate the execution status of the user applications. As mentioned before, these projects usually adopts the approach of Web Service, which follows the traditional client-server model, to offload the migrating efforts that need to be paid by clients devices and make user mobility possible. The client device, following the *thin-client* approach, can therefore be made *small* and *mobile* in the pervasive computing environment. Systems that have the faith in Web services include some typical commercial systems leaded by Microsoft's .NET architecture [18] and the Sun Microsystem's Open Net Environment (ONE) architecture [34], and research-based systems like that in the Ninja Project of UC Berkeley [33]. However, as stated in Section 2.1, these systems utilizing the centralized approach are generally less flexible and adaptive when compared with the use of mobile codes in the pervasive computing environment.

In fact, there are a few systems in some other projects do not adopt the Web service approach to address the issue of user mobility. One of them that possess the most similar vision as that in our project in pervasive computing is the **one.world** project of the University of Washington [9]. Similar to our Sparkle project, their model also separate the management of *data* and *functionality* to simplify sharing, searching and translating of data and letting the two evolve independently in the highly competitive pervasive computing market. In their model, there are three main abstractions: *Tuples*, which represents data as self-describing records and used for storage, networking and events; *Components*, which implements functionality and interacts with each others by exchanging asynchronous events; and *Environments*, which group stored data and application functionality, and even another environments (which is similar to our abstraction of *container* introduced in the later sections). These abstractions will run on a system supporting various core services, like checkpointing services, migration services, and remote event passing. Here, we would particularly focus on its migration service.

In the one.world system, it is also believed that in pervasive computing environment, migration is an essential system service in allowing applications to follow people as they move about during the course of their day and remain continuously available. This is exactly the issue of addressing user mobility. They do not use the computational-static client-server approach. Instead, they just use a simple mechanism to transfer the execution status when the user moves [8]: since the environment tree has all the required application codes and data, the underlying migration service just need to move or copy the whole environment tree to the remote site, and then restore the references pointing to the non-migratable resources. The whole execution can then be restarted.

This migration mechanism design, although simple, has some design principles deviated from that of our mobile code system. Firstly, different from our mobile code system that supports strong mobility, it only provide weak mobility support. In other words, migration is only possible at pre-defined execution points. In their case, migration can only be applied to the tasks that are pending on the communication queues. Executing tasks are not allowed to be migrated. This makes some valuable features for pervasive computing environment, e.g. *delegating execution when there are not enough resources on the devices*, impossible to be supported. Furthermore, we also doubt that whether multi-threaded executions that involves inter-thread communications (i.e. the use of `synchronized` block, methods like `wait` and `notify`, etc.) can be migrated[2]. Secondly, the one.world system adopts a eager approach on migration. It will eagerly migrate all the states that is relevant to the environment abstraction. This is different from our design principle: we believe that this eager approach will waste the overall bandwidth consumption. In view of this, our mobile code system therefore uses the *on-demand* strategies (to be described in Chapter 6 to reduce the bandwidth usage (and possibly memory consumption), thereby making the system lightweight and suitable for the pervasive computing environment.

[2]This will be fully discussed when we talk about *reactive migration* in Chapter 5.

Chapter 3

Overview of the Sparkle architecture

In the previous chapter, we have seen the degree of computational flexibility that can be provided by the mobile code/agent systems in the pervasive computing world. We have also compared our design goals and features with other similar systems. Before we describe the details of our mobile code system design, in this chapter, we would give a brief overview about the nuts and bots in the Sparkle architecture: we first describe the design motivation of our architecture, followed by a brief introduction of the main data structures and components, and the interactions between them in the Sparkle architecture. With all these background knowledges, one can more easily grasp the ideas in our mobile code designs that are presented in the later chapters.

3.1 Design motivation

Traditionally, applications are built as monolithic blocks. The problem is that they become too big to fit into small devices which have only limited resources. Thus, a device's functionality is limited by its own configuration. Furthermore, such an approch does not allow for adaptability. Developers have to write programs tailored for each of their targeted client devices.

Using web services [40] is by far the most common approach to address the problem of performing computation for small devices. It simply follows the client-server model. In that scenario, software is hosted on the server, and the client devices access the service provided by the software by sending service requests and the required data across the Internet. It falls short in cases when the data should not be moved (e.g. private data) or is too large to be transferred over a bandwidth-limited network. As described in the previous chapter, this model requires a stable Internet connection, which may not be always available in the pervasive computing environment. Moreover, this model does not support peer-to-peer interaction, a precious feature in the pervasive computing world.

In view of this, our approach to address the problem is to use a dynamic component composition. Applications are built from small functional components. These components are downloaded on demand from the network and then cached for the future use or thrown away when they are no longer useful. Applications can thus be dynamically composed at run-time. The advantage of this approach is that since every component is small and can be thrown away, functionality that can be performed in a device is no longer restricted by its configuration. Also, since the functional components are brought in at run-time, it allows applications to dynamically adapt to the client devices. If there are two components of the same functionality, the component which is more suitable for the client device or the user preference would be brought in. Moreover, since functionality is a high level abstraction, it can "move" across the device boundary, and the same functionality (but with different implementation) can be brought into the target device.

Components in our system are called *facets*. Facets are hosted on facet servers. Clients request for facets from the *Intelligent proxy servers*. These servers are responsible to return a suitable facet to client, taking into account its device configuration, the surrounding execution environment and the preferences of the user. There are also some *computation servers* for the devices to delegate the execution of facets to, just in case that they do not have enough resources to execute. The client devices interact with each other in various ways. A client may get some data from its peer, or it may ask a peer to execute a facet for it. All these can be done through the *Lightweight Mobile Code Systems* integrated in the client system. By using the mobile code system, user mobility can also be supported, in which user is allowed to move to another device to continue his working session. This can be achieved by migrating the corresponding data and execution states through the use of mobile codes.

3.2 Essential data structures

3.2.1 Facets

Facets are units of composition with two essential features: (1) each facet carries out a single *functionality* and (2) a facet has no persistent state. ***Functionality*** here can be seen as a contract with clearly specified inputs, outputs, pre-conditions and post-conditions. Facets, on the other hand, are just components that implements these functionalities. They take in inputs and give the desired output, according to the pre-conditions and post-conditions specified in the contract.

Having a single functionality makes the functional components smaller, and also simplifies the run-time composition. Since a facet has no persistent state, there are no dependencies between two calls to the same facet. This makes the component *throwable after use*, freeing up memory and resources for the facets that are currently running and to bring in other facets. In addition, clients are free to use other facets with the same functionality on the next call.

Facets, similar to the concept of *functions* in common programming languages, may call upon the functionalities provided by other facets to fulfill their contracts. ***Facet dependencies*** are the *functionalities* that a particular facet depends on. Two facets can have completely different implementations and yet achieve the same functionality. In other words, different facets may have dependencies on totally different sets of functionalities. However, As long as they stick to the same contract, the facets are *compatible* with each other[1].

At runtime, the client will send a facet request that contains a *facet specification* to the network, which includes the required functionality, runtime information and other requirements. The network will return a facet which matches that specification to the client. It is possible that each time, a different facet is returned for the same specification. Which facet is actually called can only be determined at run-time, after the Intelligent proxy server takes into account also the user preferences and the current execution environment of the client.

Facets are made up of two parts: *shadow* and *code segment*. ***Shadow*** is a description of properties of component including vendor, version, the contract of functionality it fulfills, its dependencies and its resource requirements. It is used by the network infrastructure to locate the appropriate facet for the requesting device. ***Code segment***, on the other hand, is the body of the execution code, which implements the functionality stated in the contract.

3.2.2 Container

Facets contain code segments and have a programming interface with which they communicate with each other. However, they cannot directly interact with the user. Containers act as a bridge between the user and facets. A container contains routines for the user interface and a list of facet specifications. Also, since facet have no persistent states, some state information would be stored in the container. For example, the execution status of the facet and some shared data. They must be migrated if the computation is moved to another device. More on container would be discussed in

[1]The implementation of the intelligent proxy server have extended the meaning of *compatibility* when it performs its matching service. For details, please refer to the thesis of our team member [39].

3.3 Architectural design

The overall architecture of our system is shown in Figure 3-1. As one can see from the figure, our architecture basically follows a *three-tier* design architecture: it has the *facet server* at the top end, the *client system* on the other end, and the *Intelligent proxy system at the middle*. It resembles the current web architecture, so that it inherits the distributive nature of the three-tier design, at the same time making it readily be deployed and used in the current web architecture. In this section, we are going to look into each of these components one by one.

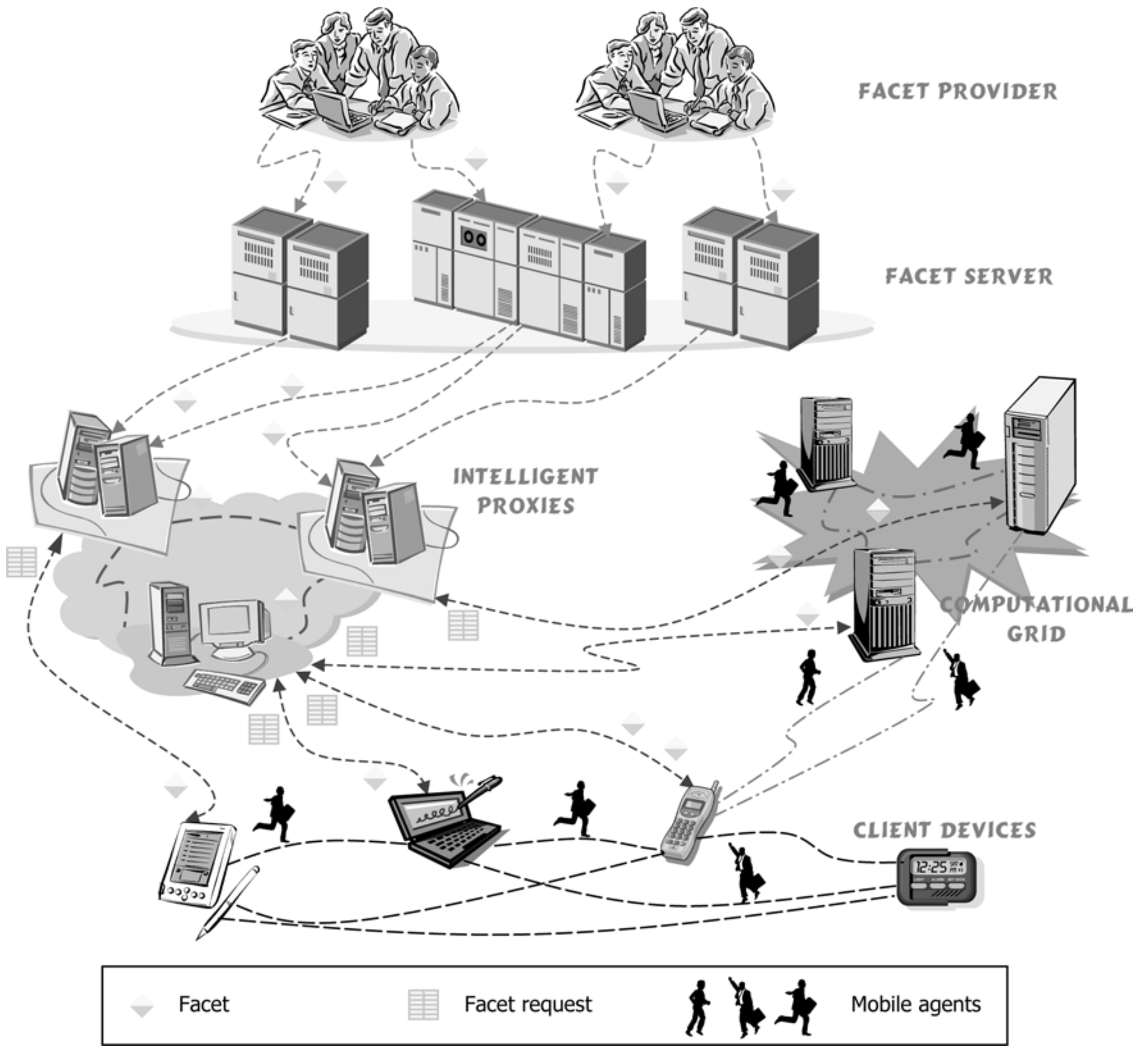

Figure 3-1: The Sparkle architecture

3.3.1 Facet server

Facet server resembles the current web servers in the web architecture. It is a place for the service providers to put their designed facets on, publish and release them to the general public. Just similar to the traditional web server design, all kinds of security or billing models can be put at the facet server to prevent the abuse of the hosted facets. Also, in cooperation with the proxy server, it will advertise the newly published facets hosted on the server by sending their associating shadows to the proxy through either a push-based or pull-based approach.

3.3.2 Client system

The client system plays a significant role in our Sparkle architecture. Since it will be installed on various client devices, it has to be small enough to fit into the devices with limited resources. It has also to be able to support state capturing and migration mechanisms for supporting various types of mobility. On top of that, it has to provide a dynamic execution environment to facets to load in, execute and then be discarded.

In particular, the client system is designed on top of a virtual machine to ensure its portability over the heterogeneous devices. The client system will accept facet specifications during the execution of container or another facet. Upon receiving the specification, it will then contact the network to request for the required facets. Once it receives the facet, it will load it in and make it available to the request invoker. Once the client system detects that the facet is no longer in use, it is also responsible to throw it away.

The client system also handles all the background housekeeping, such as locating proxies and peers over the network, keeping track of their resources usage and handling mobility through the use of the Lightweight Mobile Code System.

3.3.3 Lightweight Mobile Code System (LMCS)

This is the core part that this thesis aims to describe. As mentioned in the introduction, its aim is to provide a flexible computation model which supports all code, state and data mobility. With the LMCS, client can use a mixture of code mobility techniques in conjunction data mobility to complete their tasks.

The LMCS is adapted from mobile agent systems (MAS). The lightweight mobile agents (LMAs) are hosted on it. These LMAs do not possess any functionalities at the beginning. When they need to be used, some facets that helps it to migrate or communicate with other LMAs would be plugged into it. They will then be sent together with the tasks (abstracted by the container) they carry. On reaching the target, the facets required by the task will be executed. This facet may then call upon other facets, which are dynamcially downloaded from the network, to carry out its function.

The advantage of such type of agents is that the agents can be very small. Theoretically, mobile agents have to carry the whole code with thme when migrate. LMAs, instead need only to contain the specification of the facet that is required to carry out the task, and its associate execution state. The facet (that can be adapted to the destination host's environment) is then dynamically brought to the destination site from the nearby proxy, and continue the execution. This effectively reduce the bandwidth requirement. On the other hand, the bandwidth usage can be further reduced by using a similar state-on-demand approach, which will be fully described in Chapter 7.

With the incorporation of the LMA in our system, all the properties of mobile agents on providing computation flexibility discussed in the previous chapter can be beneficial to the pervasive computing world. This includes peer-to-peer access, disconnection tolerant, etc.

On the other hand, it can also work with the ***execution servers*** in the *computational grid*: when the resource of the client device is not enough to complete a certain task, the execution of that task would be delegated to the nearby execution servers, which are supposed to have abundant resources for task execution, by the use of LMAs.

3.3.4 Intelligent proxy system

The web would not have been so popular if the web architecture contains only web pages (server content) and browsers (the client system). It must have some sort of intelligence built into it for the clients to search their required web page over the seas of web pages. This is where search engines come in and serve this purpose. The scenario is similar in our architecture, we must have some sort of brokering entities in between client and server. When the client requests for a facet, the network should be able to return the facet that is suitable for that device configuration, as well as user preferences.

In our architecture, the *Intelligent proxy system* play this brokering role. As its name suggests, it is in fact the middle-tier sitting between client and server. It helps the facet server to receive request from the client, and returns the corresponding facets by either retrieving them from its own cache, or in the case of cache miss, fetching from the facet server. Just similar to any other proxy approaches, the use of proxy system helps to relieve the load of the facet server and the network traffic to the facet server. Also, through the use of caching and prefetching strategies tailored for the design of facet (as a code component), clients are expected to get a faster response on retrieving the facets.

To serve its brokering purpose, when returning the result to a query, it also takes into account details such as the client configuration, the current run-time status of the device, the execution environment, the user profile and preferences. Thus, it can return a facet that is best suited for the particular execution environment (adaptation) and the particular user (user customization).

Chapter 4

Background concepts on mobility

Before we go into details on describing the design of our mobile code system, there are several concepts and terms that are related to mobility issues and are commonly used in the literature needed to be described and explained. In this chapter, we will briefly go through these concepts and terms. We will first study the definitions and classifications of common terms and concepts like *states*, *mobility* and *migration*. We will also present the common approaches in dealing with data with different mobility natures. Finally, we will discuss and compare the common strategies nowadays in supporting *transparent migration*, a feature that is essential in enabling strong mobility. These definitions and concepts will be used throughout this chapter, and also in the remaining chapters in the thesis.

4.1 States

Mobile agent, by its definition, may migrate at any point in the code during execution and continues at the same point autonomously at the remote location. To do so, codes and all relevant information about the execution needs to be transferred and restored correctly at the destination site. This relevant information of execution, which is built from the execution past and describes the execute present, is in general called ***states***.

States are usually classified into *data states* and *execution states*. However, there are some diverging definitions towards this classification. One approach presented in [6] gave the following definition: ***data space (state)*** is defined to be the set of references to data entities that can be accessed by all executing entities. ***Data entities***[1] here resemble the files, the globally shared variables, and

[1]In the original paper, this is named as *resource*. However, to prevent confusion of this resource and some general

the shared objects in the object-oriented languages. ***Execution states***, on the other hand, contains private data that cannot be shared, as well as the control information related to the current execution. Examples of execution state are the values of the call stack and the instruction pointer (i.e. program counter).

In [37], a similar yet more fine-grained approach in classifying states was discussed. In their model, *data state* is used to store values that is meaningful and probably specific to the target application/functionality. In contrast, *execution state* would store the control information of the program, rather than meaningful data to the application. For clarity, They further categorized data state into *stack state*, *member state*, *resource state* and *user interface state*. Execution states were also divided into *instruction pointer state*[2] and *thread state*. The meanings of each of the above states are briefly described as follows:

- **Stack state.** This state refers to all local data in every method on the call stack. The data consists of all values of local variables (***variable stack***), as well as operands of computations on the stack (***operand stack***) up to the point of migration.

- **Member state.** This state refers to the values stored in the member variables in the agent program. Usually, it contains the dynamic query of the agent and the result data of the computation.

- **Resource state.** The resource state refers to the status of established connections to system resources that the agent owns. System resources here refers to general I/O resources (e.g. network connections, file, databases, etc.) or some static native services (e.g. printing function).

- **User Interface state.** The meaning of this state is self-explanatory: it refers to the status of the user interface. (e.g. buttons pressed, checkboxes checked, or even printed messages).

- **Instruction pointer state.** This state contains information describing how the execution can be restored on the destination host at exactly the same point of execution where migration is triggered. In particular, it is usually an ordered list of instruction pointers pointing to the executing instructions in every method frames on the ***call stack***, ordered according to their calling sequence.

execution resources like CPU, memory, network bandwidth, etc., which would also be frequently appeared in this thesis, we use the name *data entity* here instead.

[2]In the paper, it was named as "program counter state". We used the name "instruction pointer state" because the name "program counter state" may be misinterpreted as keeping just the value of program counter instead of keeping all the execution points along the execution stack. We think that "instruction pointer" is better suited for this purpose.

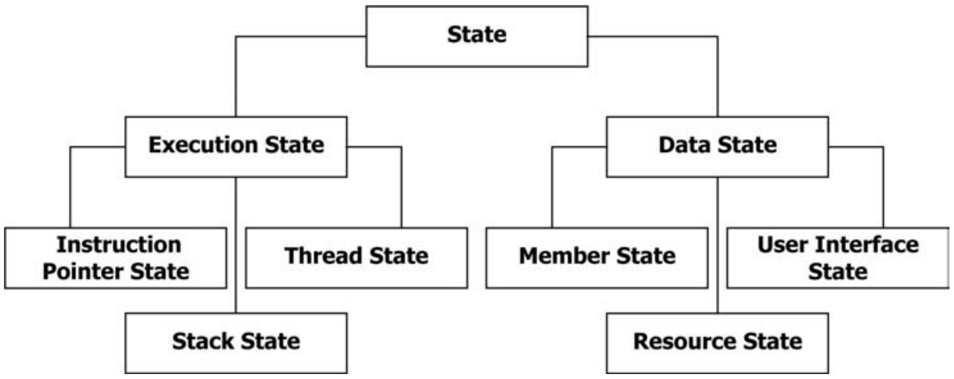

Figure 4-1: The classification of different states

- **Thread state.** This kind of state only applies to multi-threaded program, where thread is the unit of execution. In this state, descriptions about thread states (e.g. running, suspended, blocked, etc) should be carefully captured. If monitors or other synchronization constructs (e.g. semaphores, conditional variables) are involved in the execution, their status also have to be captured into this state.

This classification approach is very similar to the previous approach, except the sub-categorization in data state and execution state: In the former approach, *stack state* is part of the execution state, whereas in the latter approach, it is part of the data state.

Between these two state classification schemes, our working model would adopt the former approach for classifying states. It is because as we shall see later, some definitions on code mobility and concepts data state management mechanisms presented in [6] can be more easily established under such classification scheme. However, since in [6], what composes the data state and execution state are not clearly specified, in this thesis, we would still adopt the terms in describing various types of state used in the latter approach for clarity.

The final classification scheme is shown in figure 4-1. We would follow these definitions in the rest of our discussion. Based on these definitions, we would look into two concepts that are closely related to mobile agents in the following subsections: *mobility* and *migration*.

4.2 Mobility

Before going into details how mobility is defined, let us first define the term *execution constituents*. ***Execution constituents*** are the components that form the sequential flows of execution. Example of execution constituents include *code* and the *various types of states* described in the previous subsection. With these execution constituents, the current status of execution can be fully and uniquely described[3].

With this definition, we can now identify different forms of *mobility*. ***Mobility*** of Mobile Code Systems (MCSs) can be characterized by the execution constituents that can be migrated from one host to another [6]. To be specific, ***strong mobility*** is the ability of an MCS to allow migration of code and the complete execution state to another host. MCSs supporting strong mobility allows an execution migrated to a remote site to resume at exactly the same execution point. ***Weak mobility***, on the other hand, is the ability of an MCS to allow code transfer between multiple hosts. The code transfer can be accompanied by some initialization data, but *no migration of execution states* should be involved. It follows that for the codes running on MCSs supporting only weak mobility, some extra efforts/mechanisms are needed to be spent on resuming execution at exactly the same point and states.

4.3 Data state management schemes

One may notice that data states are not involved in the above discussions on defining mobility. It is mainly because even if data states are not migrated, by definition they can still be globally accessed by all the execution entities, and therefore allows execution to be correctly resumed at the same execution point and state. In other words, the issue of whether they can be migrated will not affect the mobility of the overall execution. What this issue would affect are in fact only the performance and the availability of the overall execution in the mobile environment.

However, the above claim is only based on the assumption that data states are able to be accessed by all the executing entities no matter where they migrate to. To support this claim, we need the so-called *data state management scheme*.

Recall that *data state* is defined to be the set of references to data entities that can be accessed by all executing entities. When we migrate the execution to another host, *data state* may have to be changed. It is because we may need to void the bindings of references to the data entities, re-establishing new bindings, or even migrate/copy some data entities to the destination host along with the execution. The choice of which of the above action to take depends on the nature of the data entities involved, the type of bindings to such data entities, as well as on the requirements posed on the usage of these data entities. How to maintain the bindings between data entities and

[3]Notice that *states* alone is not enough in describing the current status of execution. It must be used together with *codes* to describe the exact point of execution.

the migrating execution is the issue of ***data state management***.

A detailed discussion of *data state management scheme* is presented in [6]. Here we would extract some of the core idea from it.

Data entity can be modeled to have three identification attributes: a *unique identifier*, the *value of the data entity*, and its *type* that describes the structure of information encapsulated inside the data entity and its interface. Through using the interface information, the type of the data entity can also determine whether the data entity is *transferable* or *not transferable*. ***Transferable data entities*** refer to those entities that can be moved over the network. Example of these include data entity typed "stock data", or any member objects (not *reference to member objects*) in the *member state*. ***Non-transferable data entities*** is just the opposite: they cannot be migrated over the network. These can be those entities typed "printer", or any native information (e.g. files) in the *resource state*. Under this classification scheme, we may in general classify *member states* and *user interface states* as transferable states, whereas *resource states* as non-transferable states, although in some situations *resource states can be migratable*.

Instances of data entities with transferable types can be marked as ***free*** or ***fixed***. While the former is freely-migratable over different hosts, the latter is not migratable and has to be tightly bound to one host throughout its lifetime. Whether the instance of the data should be marked *free* or *fixed* depends on the nature and the usage requirement of the data entity. For example, while it is possible to transfer a huge file or the entire database to a remote site to increase the availability of data during execution, it is still undesirable because of performance issues. Also, security is another concern. Some data entity, e.g. private keys of the users, should always be marked fixed, since they should not be migrated to the other hosts throughout their lifetime.

Bindings to the data entities in general can be categorized by the identification attributes of the data entities, i.e. *by identifier*, *by value*, and *by type*. These types of bindings, to a certain extent, can reflect which part of that instance is critical to the overall application: if the instance is bound by identifier to the application, it implies that the application must need the exact copy of the data entity in order to work; whereas if the instance is only bound by type, the application may only need a data entity with compatible type in order to work. For example, if an executing code binds to a data entity with type "printer" by type, it means that wherever the execution migrates, it only needs to find a nearby data entity with type "printer" to finish its execution. If on the other hand, the executing code binds the same instance by identifier, the execution dictates the use of that same instance no matter where the execution migrates. Therefore, the type of binding established depends

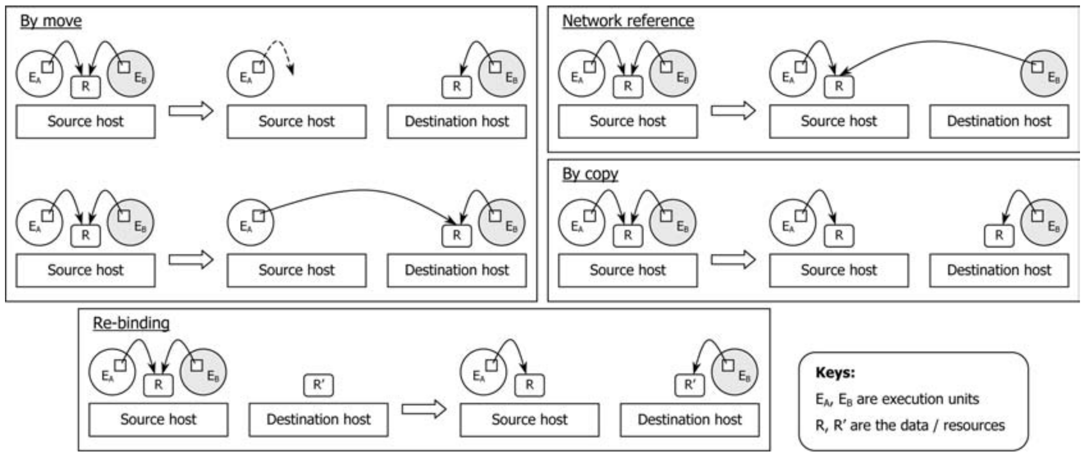

Figure 4-2: Concepts on different data state management mechanisms [6]

	Free Transferable	Fixed Transferable	Fixed Not Transferable
By identifier	By move (Network reference)	Network reference	Network reference
By Value	By copy (By move, Network reference)	By copy (Network reference)	Network reference
By Type	Re-binding (Network reference, By copy, By move)	Re-binding (Network reference, By copy)	Re-binding (Network reference)

Table 4.1: Bindings, resources and data state management mechanisms [6]

also on the nature and requirement of the application.

The above classifications bring out two dimensions of problem that must be addressed by data state management mechanisms upon migration of execution: *data entity relocation* and *binding reconfiguration* [6]. Typical mechanisms in addressing these issues involve *data copy*, *data move*, *network reference* and *re-binding*. The concepts of these mechanisms are illustrated in figure 4-2, and the mapping between the classes of problems and their corresponding solution is shown in table 4.1. For detailed discussion on this topic, readers are referred to read the corresponding section in [6].

As a final note, it should be noticed that the nature of the data entity and the type of binding is often determined by the language definition or the implementation determined by the underlying MCSs, rather than the application programmer, thereby constraining the available mechanisms exploited. For example, serializable objects in Java is free-transferable (no options in Java to specify fixed-transferable) and they are often bound by value. Therefore, object serialization becomes the dominant data-state management mechanism, which adopts the *data copy* strategy. Other types of data-state management mechanism for dealing with other types of data entities needs to be implemented by the application programmer.

4.4 Migration

Migration is one of the mechanisms used in supporting mobility. In principle, the migration mechanism would suspend the current execution, capture its associate states and transmit them to the destination host, and then resume the execution. Migration can be either *proactive* or *reactive*. In ***proactive migration***, the time and destination for migration are determined autonomously by the migrating execution unit. One applicable area of this migration is on the traditional mobile agents, in which their migration decisions are autonomous and self-responsible. In ***reactive migration***, movement of codes and states are triggered by a different execution unit that has some kind of relationship with the migrating execution unit. One classic example on the use of this is to achieve load-balancing effects by migrating the code and states of the executions to different hosts in the load-balancing systems.

As a remark, it should be noted that our mobile code system has to support both proactive and reactive migration. While the proactive migration support allows agent programmers to specify when and where the agents migrate, the reactive migration support is also necessary to grant the MCS authority to kick away agents, thereby allowing it to exercise load-balancing/resource scheduling policies. Moreover, reactive migration support is also inevitable in enabling state-capturing in multi-threaded applications. It is because different from single-threaded application, the proactive migration of an multi-threaded application is initiated by one thread only. For all the other threads, they have to migrate according to the migration signal being sent by that proactively-migrated thread. This implies reactive migration support is needed in that scenario.

Migration can also be classified according to how *transparent* their migration mechanism is from the agent programmers' point of view. Here, the ***level of transparency*** can be defined to depend on the number of execution constituents that can be transparently migrated[4]. With this, we can now give a more formal definition to transparent migration: ***totally transparent migration*** (or *strong migration* in some papers) is a migration mechanism that can transparently migrate *all execution constituents*[5] to the remote destination. The relationship "*stronger than*" or "*weaker*

[4]Although migration is often applied to support strong mobility, which takes into account only codes and execution states, it is generally beneficial to migrate also free-transferable data states. Therefore, if a mechanism can transparently migrate data states, it is said to be more transparent than those that cannot

[5]Since it may be impossible to migrate some of the data states, e.g. resource state, to the remote host, "all" here refers

than" between the migration mechanisms can also be derived directly from the level of transparency supported.

4.5 Concepts related to mobility and migration

Summarized from the previous sections, an system with strong mobility support indicates that there is a built-in mechanism in the system to save, transfer and restore the execution state of an execution. It follows that transparent migration support on these systems is not important, since even if these execution manipulation routines are not transparent to programmers, programmers can themselves easily manipulate the required execution state as they want using some predefined system routines. In these systems, agent migration can be supported directly and the autonomy of agents can be preserved.

However, on an MCS supporting only weak mobility, the above mechanisms on migrating the execution states are not available. In this scenario, there are two common approaches in handling the migration event.

Restart the execution from predefined points with initialization data

This approach follows directly from the definition of weak mobility. It is a conservative approach in addressing the migration issue, yet it is common among the Mobile Agent Systems, e.g. IBM's Aglets [15], ObjectSpace's Voyagers [31], etc. The working principle of their approach is simple: the programming framework utilizes the notification event-based model, and provides some callback methods for the programmer to implement. Upon receiving some migration events (e.g. `onArrival`, `onDispatching` in Aglets), these callback methods would be executed accordingly. Through this event based model, agent programmers can now save the relevant states when the agent is dispatching, and recover the corresponding states when the agent arrives the destination host.

However, the above scheme exhibits the following problems:

- In the system supporting only weak mobility, execution states such as instruction pointers and stack data are not available. Moreover, in the scheme stated above, the callback methods has no way to know which method the agent is executing when the migration is triggered, not to say saving their associate values of instruction pointers and stack data. In other words,

to all execution constituents except fixed or not transferable data states.

the state saved cannot fully describe the current execution status, and therefore, precise agent migration cannot be achieved.

- A direct impact from the above scenario is that the *autonomy* of the agent *decreases*.
- *Reactive migration* is also not possible due to the same reason: an external execution entity has no way to know which method the agent is executing, not to mention saving execution states and control/migrate the execution.
- The consequence of the above scenario is even worse: if there are some *side effects*[6] during execution, e.g. the execution has modified some values inside globally shared object, it is not easy, if not impossible, for a migrated execution to recover at exactly the same point. In other words, the execution may not be correct any more.
- In other words, in order for the agent to work correctly, the agent programmer has to consider different scenario associated with different forms of states where migration is possibly triggered. This, together with the manual state saving steps, further burdens the workload of the programmers.

Simulating strong mobility on top of weak mobility

It can be seen from the above approach that weak mobility cannot satisfy the need of mobile agents, which exhibits automatic and highly-mobile nature. In view of this, there have been several attempts to simulate strong mobility on top of weak mobility. They can in general be classified into two different categories.

- **Using non-transparent migration.** Similar to the above approaches, agent programmers need to take care about the state-saving and migration details. However, these approaches differ from the above approaches in that: instead of putting the state-saving statements inside the call-back method (which is considered to be an external execution entity), the agent programmers need to put the statements inside the agent execution code. This allows them to save some information about the execution state of the agent, e.g. data on *variable stack* and *operand stack*, and also some information about the instruction pointers, which in turn allows them to recover the execution state at the exact execution points.

[6]*Side effects* here refer to the modification of non-local variables, e.g. modification to global variables, content of files, content on the screen, etc.

However, there are two major drawbacks in adopting this migration scheme:

  - This approach again burdens the agent programmers. The implementation of the state-saving mechanism of the agent would be error-prone if the flow of the program is not carefully designed and analyzed. The resulting mechanism may therefore involves very complicated and tedious logic.
  - Since the state-saving statements are inserted directly inside the agent's code, the resulting code may become messy, which makes the execution flow of the program unclear. As the application grows large, programmers may eventually find it very difficult to analyze the flow of the program and understand its behaviour.

- **Using transparent migration.** This approach is similar to the first approach, except that the code instrument process is made transparent to the programmer. This can be achieved by adding a pre-processor or modifying the underlying system. From the viewpoint of the agent programmers, it is an important issue since it allows the programmers to write mobile agent applications in the same way as writing non-mobile agent applications [24]. It substantially supports the agent programmers to understand program behaviour, since in the programmers' view points, no state-migration statements will be added.

Making the agent migration transparent to programmers can surely aid them to write agent programs, especially to those agent programs written for MCSs implemented on systems supporting only weak mobility, e.g. Java [1]. There are now several different strategies in making agent migration transparent, each of which have their relative strengths and weaknesses. We will discuss more on this issue in the next section.

4.6 Transparent migration on weak mobility systems

Java [1] has been very popular in developing distributed applications with mobile codes and agents. It is mainly because Java has several benefits in implementing these systems:

- **Machine independent.** Java's *machine independent byte-code interpretation*, together with its *virtual machine technologies,* allow the implemented agent application to be executed on heterogeneous platforms. The *portability* of the system hereby introduced is precious to Mobile Code Systems, which are supposed to work on a highly diverse types of platforms.

- **Dynamic class loading support.** Java provides *dynamic class loading* support, in which the bytecode of an application can be dynamically transferred over the network and loaded during code execution. This allows the code of the agent application be freely migratable to wherever the agent migrates, thereby easing programmers' effort on handling the code transfering issue.

- **Object serialization.** Java provides also *object serialization* facilities to transparently convert the *member states* of an object to byte stream and transfer to the destination host. This also eases programmers' effort on explicitly transferring all the relevant data states to the destination host.

Because of the above reasons, many tools and systems for mobile agents are almost completely implemented in Java, Some well-known examples on such systems are Aglets, Voyagers, Grasshopper, etc. However, since *Java security policy* forbids the dynamic inspection of the execution stack and all other relevant execution status, the execution states cannot be easily and directly captured in Java. In other words, Java supports only weak mobility. The direct implication of this is that all the mobile agent system mentioned above the migration exhibits the problems on weak mobility system stated in Section 4.5.

A better way to deal with the migration problem is to simulate strong mobility on top of the weak mobility system. As discussed before, there are in general two approaches: non-transparent migration and transparent migration. However, since non-transparent migration would place unnecessary burdens on the programmer (as stated in Section 4.5), in the following section we would put our focus on various strategies on supporting transparent migration. In the rest of our discussion, we would take Java, a language that is commonly used in implementing mobile agent systems, as an example to illustrate the working mechanisms behind these strategies. Similar strategies should be able to apply on other programming framework as well.

4.6.1 Common strategies on transparent migration support in Java

In general, there are four different strategies applying to different constructs in the Java programming framework that would be used to support transparent migration. They are briefly described as follows:

- **Source code instrumentation.** In this scheme, the structure of the program would be analyzed, and the state-saving and state-resuming programming constructs would be added into the source code of the target program accordingly. The *preprocessed* source code would then be sent to the compiler for generating byte-codes that *virtually* supports strong mobility. Examples of systems adopting this strategy includes the agent system in *WASP* project developed at the Darmstadt University of Technology [32], and that in *JavaGo* project developed at the University of Tokyo [27].

- **Byte code instrumentation.** This scheme is similar to the source code instrumentation scheme, except that the whole instrumentation strategy is done at byte-code level instead of source-code level. The *Brakes* project developed at Katholieke Universiteit Leuven [38] and the *JavaGoX* project developed at the University of Tokyo [24] follow this strategy.

- **Virtual machine modification.** The working principle of this approach is based on the fact that all the execution information are available in the virtual machine layer. Therefore, the standard Java Virtual Machine can be modified to support strong mobility, by adding some non-standard interfaces to bring the corresponding execution states from the virtual machine to the code running on top of it. Code resuming can be done in a similar manner. The *NOMADS agent system* [35] is an example of adopting the above scheme to perform transparent agent migration. Some other non-agent-oriented system, e.g. JESSICA [16], Sirac [26], Ara [22], etc., also adopts this approach on implementing transparent thread/process migration to achieve load-balancing purpose.

- **Using Java Platform Debugger Architecture (JPDA).** JPDA is part of the virtual machine specification and normally used to develop debuggers for Java applications. Through using JPDA, some runtime information, e.g. running threads, call stack, program counter, etc., can be directly retrieved. However, since there are not enough functions to support the direct modification of these execution states in JPDA, i.e. only allows inspection but not modification, the restoration of execution states have to be aided by the source-code or byte-code instrumentation mechanisms described above. The *CIA System* developed at University of Ulm [37] is one of the examples of adopting this approach on transparent agent migration. *M-JavaMPI* developed at the University of Hong Kong [17] adopts a similar approach in enabling Java process migration.

4.6.2 Strategies Comparisons

Virtual machine modification

The *virtual machine modification* approach clearly provides the highest level of migration transparency to the agent programmer: no code overheads would be introduced; no limitations on the programmers' coding style would be placed; and there is no significant performance degradation compared with other schemes (because the information are directly available inside the virtual machine). However, since the virtual machine modified no longer sticks to the Java standards, it therefore requires the mobile applications to run only on the modified virtual machines. The portability of the overall system, which is one of the major concerns in designing mobile agent system in heterogeneous environment, is therefore compromised. After all, modifying virtual machine is not a trivial task, and is thus not very suitable for implementing general mobile agent systems.

Compared with virtual machine modification, since all the other three approaches are dealing with different parts *above* the virtual machine layer, the portability of these approaches are preserved.

Source code instrumentation

The *source code instrumentation* strategy adds in state-saving and state-resuming codes inside the source codes of the program. Some code overheads would then be introduced in both the resulting source code and bytecode, thereby slowing down the execution speed and increasing the bandwidth usage during migration. Also, this approach is feasible only when the source code of the application is available. In other words, it is impossible to save the state during execution of library codes or methods without source codes. Thirdly, because of the inherent nature of the Java language, there are some limitations on the execution locations where the migration event can occur, e.g. the migration event cannot be triggered when the current execution is still in an object constructor, because it is impossible to transfer an object when it is only halfway constructed. Last but not least, since an additional parameter (the execution state) has to be added into the method signature, some methods that must keep their signature intact (e.g. some event callback methods like `actionPerformed`) cannot have their states saved.

In spite of having so much disadvantages, the source code instrumentation approach has its own strengths. Since the variable declarations and scoping rules can be applied at the source code level, the semantics of the code, e.g. the information about the structure of variables, are easier to be preserved when compared with byte-code level instrumentation. Also, since the instrumentation is done at the source code level, programmers, to a certain extent, can understand the internal workings of the resulting code, which aids to debug their agent programs. Finally, since the source code instrumentation is done before compilation of code, the code can be compiled with some specific options provided by the compiler, e.g. compiling with debugging option turning on, thereby enabling some standard debugging information to be inserted to the compiled byte-code.

Byte code instrumentation

The *byte code instrumentation* scheme, similar to source code instrumentation, would insert state-saving and state-resuming codes into the byte-code of the program. This again would introduce some code overheads, but smaller than those introduced in the source-code instrumentation scheme. It is because some instructions that are beneficial to state-saving and state-restoring, e.g. `goto` for restoring *instruction pointer state*, are available only at byte-code level. The resulting code can therefore be smaller in size and can run more efficiently. Secondly, Java codes are usually deployed in their byte-code forms, and their associate source code is not always available. Byte code instrumentation thus has an obvious advantage in such scenario: the source code of the program is not required for transparent mobility support. Finally, byte-code instrumentation allows codes to be instrumented at runtime (or more precisely, the *class-loading* time) by using a customized class loader. In that case, the bytecode of the program can be kept intact. It would be ideal in some situations where transparent migration support is an optional feature, so that the original code can be reused when migration is not required. Furthermore, since no code overheads would be introduced before instrumentation is done, some code transmission bandwidth would be saved, although at an expense of wasting runtime computation cycles.

The major drawback of the *byte-code instrumentation* scheme is its difficulties in designing the transformation scheme [24], which are raised mainly because of the constraints posted by the Java Architecture. Firstly, the transformed byte-code must pass a byte-code verifier, in which the consistency of types of the variables declared are checked. Secondly, it is difficult to know the set of values to be saved and restored. In byte-code level, values are passed by frame variables and the operand stack, and there are neither variable declarations or scoping rules. In other words, a type system of Java bytecode has to be used to statically Analise the code structure in order to correctly modify the program. However, even if so, it will still be difficult and tedious, if not impossible, to design the instrumentation scheme if the logic behind capturing and resuming the execution state is complicated, e.g. the thread state saving and resuming mechanism to be discussed in Chapter 5. Lastly, since the byte-code instrumentation scheme would affect the correctness of the debugging information inserted to the byte-code by the compiler [24], e.g. line number table, some extra efforts may be paid to enable these features.

Using Java Platform Debugger Architecture

As mentioned before, *Java Platform Debugger Architecture*(JPDA) is part of the Java virtual machine specification, and should be implemented by every Java virtual machines. Through using the *Java Debugger Interface*(JDI), a component of JPDA, the information about the execution state can be directly retrieved without any code instrumentations, thereby producing almost no code overheads on execution state capturing (except the debugging information inserted).

Despite the simplicity and the cleanliness of the approach, there are several problems associated with it. First of all, since this approach relies on the use of debugging information, all the codes have to be compiled with debugging option turning on. For the byte-codes without debugging information inserted, which are the majority part among the existing codes deployed, the above mechanism cannot be used. Secondly, not all the information about the execution state are readily available from JPDA, e.g. the status of the *operand stack*. Furthermore, there are not enough functions to support the direct modification of execution states (e.g. program counter) in JPDA. Therefore, in these scenarios, the code instrumentation or the VM modification schemes would have to be used in co-operation. This implies that the problems for the other approaches stated before may also exist in this approach. Last but not least, when the code is run in debugging mode, the Just-In-Time(JIT) compiler inside the VM is forced to be disabled. It follows that this approach would suffer from great reduction in execution efficiency when compared to codes running in normal JIT mode [37].

The comparison of the above four strategies can be summarized in the following table:

	Virtual machine	**Source code**	**Byte code**	**JPDA**
Portability	No	Yes	Yes	Yes
Performance	Good	Reasonable	Reasonable	Bad
Code overhead	None	Moderate	Little	Little
Method signature modification	No	Depends on implementation	Depends on implementation	No
Implementation Difficulty	Difficult	Moderate	Difficult	Easy
Special requirement	N/A	Source code should be available	N/A	Byte-codes compiled in debug mode

Table 4.2: Comparisons of different strategies in supporting transparent migration

Chapter 5

Portable support for transparent thread migration

As we have discussed from the previous chapters, there are several important issues on designing mobile code system. Two among them are the portability of the system in heterogeneous environment and the system support for transparent migration. With portability, mobile code systems can be installed on a wider ranges of devices and platforms, thereby allowing mobile codes and agents to operate on the globally distributed and heterogeneous environment and provide more flexibility on general computation. On the other hand, with transparent migration introduced, agent programmers can put their focus on their agent designs, rather than on the internal workings of the migration process, while at the same time enjoying the benefits brought by *strong mobility*. Furthermore, if *reactive migration* is introduced, it is even possible for mobile agent systems to exercise control on the activity of agent according to their own resource control and security policies. Also, as one will see later, it is also an essential feature in supporting thread-state migration.

However, to support transparent agent migration on an MCS supporting only weak mobility (e.g. MCS supporting only Java), *complete execution state migration* is a difficult issue. Recall from Section 4.1 that execution state is composed of three main components: instruction pointer state, stack state and thread state. To our best knowledge, almost all the existing transparent migration approaches on weak mobility systems discussed in Section 4.6 only dealt with the migration of instruction pointer state and stack state (except VM-modification approach, which is *not portable*). The migration of thread state, which is also an essential element in supporting multi-tasking in mobile environment, was often ignored or was identified as the future work. In other words, there are currently *no* portable schemes on supporting transparent thread migration on weak mobility systems, which implies that the issues on migrating mobile code is still not yet fully solved.

In this chapter, we will first study the main technical difficulties that we have to overcome in order to support transparent thread state migration. After that, we will briefly go through the current portable schemes on supporting execution state migrations, and discuss why these schemes fail to support thread state migration. Then, we will describe our source code instrumentation strategy, analyzing the possible execution scenarios that can arise in multithreaded applications and how our scheme will deal with that. Finally, the limitations of our instrumentation scheme would be discussed.

5.1 Background and Objective

Multi-threading is a well-known and popular technique used in desktop applications. With multi-threading, the application can perform several functionalities in parallel. The programmers can also easily manage tasks with different semantics and goals to work on the same data space, thereby relieving their programming efforts. Furthermore, the parallelism provided by multi-threading introduces more scheduling flexibility to the scheduling policy of the system, which can therefore make better use of the available resources.

Multi-threading support is equally important in mobile agent systems. Although mobile agents are usually used to accomplish one particular goal, they are not necessarily singly threaded. In fact, there are many scenarios among the mobile agent applications where multi-threading techniques are useful. For example, an information-collecting agent may need to collect information from several sources, e.g. the MCS hosting the agent, the other agents hosting on the same MCS, or some other neighbour MCSs, etc. In handling information coming from these multiple sources, multi-threading is by far the most suitable technique: by making the data storage area of collected information be guarded by the critical section for ensuring consistency, the agent can now create new working threads upon discovering new information source, and make them process the information retrieved from these sources independently. Therefore, with multi-threading, agent applications can be made more flexible and powerful.

We have discussed in the previous section that *strong mobility* support is a natural design for mobile agent system, and transparent migration is the best way of simulating strong mobility on weak mobility systems. However, incorporating multi-threading support with transparent migration on weak mobility system is not a trivial task.

- As mentioned before, the migration models between a single-threaded agent and that of a multi-threaded one are totally different. While the agent designer can consider adopting *either* proactive *or* reactive migration for the case of single-threaded agents, the designer has to consider *both types of migration* for the case of multi-threaded agents. This is because different from single-threaded agent, the migration of the multi-threaded one is initiated by one thread only (proactive migration). For the rest of the threads, they have to migrate upon receiving the migration signal (reactive migration).

- On weak mobility systems, there are only *few* or even *no* information available for execution states. It is often the case that information about thread states, being one of the execution states, are not fully available to the agent programmers.

To justify the second claim, let us again take Java as an example. Java does not provide any programming support on dumping the full thread states. The only available information about a thread state according to Java API are methods `isAlive()` and `isInterrupted()`, which check whether the thread is still running and has been interrupted respectively. Nevertheless, these status information are not enough to capture and resume the execution state. For example, through using these calls, it is still not clear whether the thread is currently running, suspending or waiting on locks. Even if these information are available, i.e. suppose we know that the thread is now waiting on some synchronization locks, Java does not provide any ways for us to know exactly which locks the thread is holding. Furthermore, the state reestablishing problems remain, as Java does not provide any means for programmers to set the threads' states.

5.1.1 The need of thread state capturing

One may then wonder if there is a need to capture, transfer and migrate thread states in a transparent thread migration scenario. In fact, it is necessary: *a migrating execution without migrating its associate thread states is simply identical to a multi-threaded execution without synchronization points*. Consider the following scenario: a thread in the critical section of a multi-threaded program triggers the migration signal, and all the threads (together with the other threads waiting outside the critical section) migrate to the destination without migrating their associate thread states. At destination, the system would resume the threads in random orders. And since there is no thread state descriptions available, there is a chance that other threads that were waiting outside the critical section resume execution before the one inside, and therefore they are allowed to tamper the critical section. The execution consistency is thereby violated.

To tackle with this state-migration problem, one may immediately think of a trivial approach: to capture the execution state when all the threads are in the same pre-defined state, e.g. *running state*[1]. If this requirement is met, we do not need to store the associate thread state, since they are now all in the same state. The approach may seem to be feasible at first glance. However, there are several problems associated with it:

- We do not know whether there exists such a snapshot of all running threads having the same thread state throughout the whole execution. For example, if the synchronization blocks in a program are very closely placed, it may be difficult or even impossible for all the threads to be in running states at the same time: most of them will always be waiting for the key of the synchronization lock. Also, the difficulty in finding such a snapshot increases as the number of executing threads increases.

- We cannot get precise information and control on all the threads at programmers' level. It is difficult for a thread to know if the other threads are in running state or not. Furthermore, even if a thread knows that all other threads are in running state at time t, it may not be able to block the other threads for migration at exactly time t. Instead, it may be able to block them at time $t + \Delta t$. However, at that time, the other threads may already be in some other states.

  As a side note, precise information and control are only available at the virtual machine level. However, modifying the virtual machine would make the approach not portable.

- Even if all the threads are guaranteed to be in running state, the approach is still incorrect. It should be noticed that even though all threads in running state, there may be cases that some of them are executing inside some distinct monitors, while the other ones are executing outside the monitor. In that case, consider the following scenario: suppose at the above moment, all the execution states of the threads (except thread states) are captured and transferred to the destination site. At the destination site, the system would resume the threads in random orders. Suppose the thread that originally executes outside the monitor got resumed first. After some execution, this thread reaches a monitor that was originally owned by some other

[1]Notice that it is not possible for all the threads to be in *waiting state* at the same time. The only possible choices here are *running state* or *blocking state*.

threads. However, at this moment, the thread that owns the monitor still does not get resumed yet. Therefore, the executing thread got the ownership of the monitor, which makes the execution inconsistent.

With the above claims, we have shown that for multi-threaded programs, it is impossible *not* to transfer any thread states while still maintaining the consistency of the program using any *portable* approaches. It is clear that we need to find a portable mechanism to save, transfer and resume thread states.

5.1.2 Current Literature

The *thread state migration* problem, however, is not clearly addressed in the current literature. Most of the recent researches aim only at migrating the instruction pointer states and stack states, while few efforts have been paid on migrating thread states: migrating thread state is usually classified as future work in these projects.

The byte-code instrumentation approach presented in [38] claimed that their scheme supports multi-threading and concurrent execution. In their approach, they use an abstraction called *task*, which encapsulates a Java thread that is used to execute that task. As such, a task's execution state is in fact the same as the execution state of a Java thread in which the task is running. Also, they used a *task scheduler* which overrides the scheduling support of JVM and schedules the execution and handles the context switching of tasks. This task scheduler would, however, keep one Java thread running at any time. Therefore, their approaches just address the migration of stack states and instruction pointer states of different tasks, while ignoring the migration of the thread states. It was not clearly described in their paper whether their middleware system would provide supports on transparently capturing and resuming thread states and, if so, their approach toward this problem. The thread state migration problem therefore remains unsolved.

5.1.3 Our objective and working environment

Our research objective is therefore to design a portable approach in transparently migrating the thread state, together with other execution states, of an executing multi-threaded program. It involves two main problems: how to capture the thread states into a serializable form for state migration, and how to resume the execution in an exactly same state as that before migration using that transferred state. We limit our scope of discussion on the four primary inter-thread communication constructs, since they represent the most typical ways in which the threads interact:

- **Synchronization blocks.** They are used to protect the critical section, preventing them from being access by more than one thread concurrently, which may otherwise give rise to *race conditions*.
- **The `wait` operation.** It is used by the running thread to suspend its own execution (on a certain object) to fulfill certain execution orders in a multi-threaded program. It would be suspended until the other thread invokes the `notify`/`notifyAll` operation on the same object.
- **The `notify`/`notifyAll` operations.** It is the reverse operation of the `wait` operation, in which it wakes up a thread suspending on a particular object. It is also used to fulfill certain execution orders in a multi-threaded program.
- **The `join` operation.** This operation can be used for the caller thread to wait for the completion of the called thread.

For the rest of discussion, we would use Java, a popular language used in implementing MCSs, as our programming language basis. We would follow Java's synchronization model in our design, although the discussed design strategy can in general be applied to many other object-oriented based programming model adopting the same synchronization model as Java, such as C#.

5.2 Our instrumentation approach

In our design, we have adopted the *source code instrumentation* approach for supporting transparent thread migration. It is mainly because we want to keep the instrumentation scheme simple and direct, making rooms for future optimizations if our proposed scheme works. Source code instrumentation is the best candidate in fulfilling this requirement. It allows us to easily insert codes that fit our current architecture with the exact semantics that we want to express. In addition, as mentioned before in Section 4.6.2, it provides better debugging support to us and agent programmers when compared to other code instrumentation schemes.

On the other hand, the drawbacks in adopting the source code instrumentation scheme mentioned in Section 4.6.2 have only minimal effect in our architecture. In our system, we need not worry about the absence of source code, as all programmers that needs to use our facet model have to build their facets from ground up. Also, by adjusting the facet model design, we can also hide the constraint being placed on instrumentation locations from the agent programmers. For example, since it is unnatural for facets to create new instances, we limit the use of the constructors in facets in our facet model, at the same time avoiding the problem of placing migration statement inside the constructor.

There is yet another problem of adopting the source code instrumentation scheme: it needs to modify the signature of the method in order to pass the transferred state information to the target method for reestablishment of states. This feature mainly makes two kinds of methods with fixed signatures not modifiable: (1) Methods that are being analyzed by other methods using the *Java reflection API*, in which the method signature may be critical in changing the program's behaviour; (2) Call-back methods that is used in event-driven programming model. However, we believe that mobile agents can still achieve most of their functionalities even if the reflection API is not used. Also, the uses of reflection API are generally discouraged in Java because programs without using it would be easier to debug and maintain [7]. On the other hand, we would try not to modify the signatures of the call-back functions in our design. Some other mechanisms are adopted instead, as described in the later sections.

Our transparent thread migration scheme was designed based on the source code instrumentation strategy used in JavaGo [27] developed by the AMO Project of the Tokyo University. Through their strategy, the stack state and the instruction pointer state could be saved at the time of migration by means of *exception-handling mechanism*. The resulting performance overhead was therefore low, which was claimed to be about 20% in most cases. This low overhead introduced makes source code instrumentation a reasonable approach to be used in supporting transparent migration.

5.3 Overview of the instrumentation scheme in JavaGo

JavaGo adopts a source code instrumentation scheme. However, it would not modify every lines of the source code provided. Instead, it provides three additional primitives for agent programmers to describe their own migration policy/mechanism. According to these primitives, JavaGo would instrument the corresponding source code accordingly.

5.3.1 The migration primitives

JavaGo provides three additional primitives for agent programmers to describe their migration scheme flexibly: *go*, *undock* and *migratory*. Their uses are briefly described as follows:

Initiate migration using `go` primitive

Migration takes place by executing the `go` method. The `go` method would take an argument describing the address of the destination MCS. When the `go` method is called, the execution state[2] and the objects (member states) will be transmitted to the destination MCS. There, the execution would continue right after the `go` statement.

For example, for the following lines of pseudo-codes:

```
print (‘‘At source’’);
go (destination address);
print (‘‘At destination’’);
```

The string `At source` would be printed at the source host, whereas the string `At destination` would be printed at the destination host.

5.3.2 Controlling migration effects using `undock` primitive

An `undock` statement block serves as a *marker* that specifies the range of the area to be migrated in the execution stack. The brackets of the `undock` block restrict the effect of a `go` statement to the enclosed statements, preventing the effects from going beyond them. In particular, when the `go` statement inside the `undock` statement block is executed, the rest of the statements after the `go` statement in the `undock` block would resume its execution at the destination MCS, while the remaining statements after the `undock` block would be executed at the source MCS at the same time.

To illustrate the idea, let us look at the following lines of pseudo-codes:

```
undock {
   go (destination address);
   print (‘‘At destination’’);
}
print (‘‘At source’’);
```

[2] In the original version of JavaGo, only *instruction pointer state* and *stack state* are migrated. Therefore, execution state here refers to only the above two states. The concept of `go` primitive, however, can be extended to be used on migrating thread states as well.

Same as the previous example, the string At source would be printed at the source MCS, whereas the string At destination would be printed at the destination MCS. It is because only the codes inside the `undock` block would be migrated to the destination MCS. The last statement is therefore not “migrated”.

It should be noticed that the statements after the `undock` block *should not* modify the member states that would be used by the statements inside the `undock` block, as this would give rise to some unpredictable execution results. It is because the statements after the `undock` block would be executed at the same time when the code inside the `undock` block is scheduled to migrate. Since the exact time of when the code would be migrated is unknown, if there are some state changes in “shared” data at this moment, these changes would be brought to the destination, making the execution results of the code unpredictable.

However, in most agent applications, there should not be any statements executing after the `undock` statement block. It is because mobile agents, by their definitions, should not have any residue executions on the source MCS once they migrate. It follows that the whole agent execution should be enclosed by the `undock` block.

Declaring migrating methods using `migratory` primitive

If there is a possibility that migration takes place inside a method, programmers must put some special markers to that method indicating such possibility. This special marker is the `migratory` modifier for methods. In other words, the *`go` statement* and the *invocation of another migratory method* must be written in either a `migratory` method, or must be enclosed within an `undock` block. By doing so, not only that the resulting program would be easier to read and maintain, the instrumentation scheme can also get an overall idea on where to instrument the code.

5.3.3 The instrumentation scheme of JavaGo

The unit of instrumentation in JavaGo is a *method*. A method is instrumented in such a way that its instruction pointer states and stack states are stored inside a serializable Java objects. The `undock` block and `migratory` modifier described in the previous section determines if the enclosing codes should be instrumented or not. In the following subsection, we will briefly describe the instrumentation schemes of JavaGo on how two important steps in state migration: *state saving* and *state resuming* are implemented. As a reminder, it should be noticed that when we talk about the execution state with respect to the original version of JavaGo, we refer to only the instruction pointer state and the stack state, but not the thread state.

The overall transformation scheme

In JavaGo, a method is transformed in such a way that:

1. All the variable declarations are elevated and moved to the top of the method, even if the variables are declared in different scope. Variables with same names but in different scopes will be resolved and given different names during this elevation stage. This stage is needed because statements will be reordered and duplicated during the instrumentation. Elevating the variable declarations avoids the difficulties of code reordering and duplication.

2. A special variable `migEntryPoint`, which is used to save the *simulated* execution point, is also declared.

3. Expressions with *migratory effects* are split. Consider the following example:

   ```
   y = a[x] + f(x) + b[x];
   ```

   In the above example, f is a migratory method. If the method containing this statement is to be migrated after f is called, the result of $a[x] + f(x)$ must be migrated to the destination, where the sum of the result and $b[x]$ is assigned to y. However, the intermediate results of $a[x] + f(x)$ is on the *operand stack* and does not save into any local variables. To deal with this, the statement with *migratory effects* is decomposed into a sequence of atomic operations:

   ```
   tmp1 = a[x];
   tmp2 = f(x);
   tmp3 = b[x];
   y = tmp1 + tmp2 + tmp3;
   ```

   This transformation guarantees that we can avoid resumption from within an expression.

4. It captures the `NotifyGone` exception, which is thrown only when a migration signal is received. All local variable states would be saved in the handler of the exception.

5. The signature of the method is modified. An extra parameter is added for passing in the saved execution state. Under ordinary execution of code, the value of this extra parameter is always `null`.

6. State-resuming statements are also added in between blocks. Executed statements are skipped through using the *switch-case* statements.

On top of that, a state object is defined for each instrumented method, which is used to store the value of all local variables associated with that method, including `migEntryPoint`.

Saving the execution state

During the ordinary execution of the method, the current execution point is continuously saved into the variable `migEntryPoint`. When migration is triggered, a `NotifyGone` exception would be raised and capture by the method. In the exception handler, the values of the local variables would be saved into the relative state object. This state object, or more properly *stack frame*, would then be inserted into the head of a *stack frame list*, which stores all the stack frames in their calling sequence. After that, the same exception is re-thrown so as to propagate the execution to the caller of the current method. The above process repeats until the exception is captured by the `undock` block, which is expected to be at the bottom of the *call stack*. There, the stack frame list would be serialized and transmitted to the destination MCS.

Resuming the execution

When the stack frame list is received at the destination MCS, the state of the execution stack would then be established by calling the method at the bottom of the stack (i.e. the contents inside the `undock` block) with the reference to the stack frame list passed in. The corresponding stack frame for that method would then be used to restore the values of the local variables in the method. This can be done by putting an extra code fragment at the beginning of the method, which would be executed only after migration takes place. Since `migEntryPoint` is part of the execution state saved, the codes that are already executed on the source MCS can be skipped by using this value and the *switch-case* construct to simulate the effect of `goto` instruction with low overhead. The detailed instrumentation scheme on this is described in [27].

5.3.4 A small point to note

As one may have noticed, since JavaGo utilizes the Java exception model to capture the instruction pointer states and stack states, the state saving is said to be *on-demand*, in which state-saving is performed whenever migration signal is received. Another possible state saving approach would be *regular state saving*, in which the states are saved into the state object for every fixed time-interval or at some particular execution checkpoints along code execution. The latter approach has an advantage that state information can be migrated immediately when the migration signal is triggered. However, it would introduce a lot of computation and memory overheads because of the checkpointing process. Therefore, *on-demand state saving* is the common approach to deal with the state-saving problem.

5.4 Overview of our code instrumentation scheme

So far, we have seen how JavaGo performs its execution-state migration mechanism. However, as mentioned before, thread state handling and migrations are not discussed nor implemented in the JavaGo approach. Therefore, in this following sections, we will have a detailed look on how thread migration can be implemented using source-code instrumentation.

Our source code instrumentation scheme is basically based on the instrumentation framework implemented by JavaGo, which was discussed in the previous section. On top of that framework, some additional data structures, assumptions and code instrumentation schemes are added to support thread state migration. In this section, an overview of these issues would be given.

5.4.1 Data model

In our 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, and the activities of the threads are centrally co-ordinated by the 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 in flag value, all the threads would know that the migration event is triggered. Similar approaches apply to other shared states among threads, as illustrated in the later sections. As one will see later, this shared object plays a critical role in our source code instrumentation scheme. In our implementation, this shared object is in fact the *container* in our Sparkle architecture.

5.4.2 Core concept on our design

In weak mobility systems, extracting the exact execution state from the underlying system while retaining the portability of it seems to be difficult. Like most of the other code instrumentation approaches that support instruction pointer state migration, when the associate state information is not available, we extract it through the way of *execution simulation*. For example, in dealing with instruction pointer migration, JavaGo explicitly inserts some codes into the statements setting values of `migEntryPoint`, which is in fact an action *simulation* of the instruction pointer. We adopt a similar approach in dealing with the problem of saving thread states: before each of these inter-thread communication constructs, we insert some codes to *simulate* what would happen when these constructs are really executed. All the relevant thread state information during the simulation can therefore be put into a serializable state object. On the other hand, for resuming thread states, we must also ensure that the threads are resumed in the exact ways the captured thread state describe. For example, if a thread is inside the synchronization block during migration in the source MCS, this thread should also be executed in the *same block* once the execution is resumed at the destination MCS. As one would see later, this involves the use of some existing Java API, e.g. `wait` and `notify`.

5.4.3 Chapter organization

The rest of the chapter is divided into five main parts. Section 5.5 describes the new problems introduced on the existing migration schemes when the code execution changes from single-threaded to multi-threaded, and the way we modify the scheme in dealing with these problems. From Section 5.6 on-wards, we will describe our instrumentation schemes on migrating thread states: in Section 5.6, we will discuss how to deal with the states associated with synchronization locks. From Section 5.7 to Section 5.8, the way on dealing with states after invoking method calls `wait()`, `notify()`/`notifyAll()`, and `join()` would be discussed respectively. And finally in Section 5.10, we would discuss the limitations on our code instrumentation scheme and the possible workarounds.

5.5 On supporting multi-threaded programs

Before exploring the details on manipulating thread states, there are still two main problems that remain to be solved if we want to support transparent migration on multi-threaded programs rather than the traditional single-threaded ones. The first one, as mentioned in previous sections, is the support on *reactive migration*. To address this issue, we use a simple approach on supporting *coarse-grained* reactive migration. Details of this would be discussed in Section 5.5.1. Another problem is raised from the fact that upon migration, the underlying executing threads, even if they are used to execute the same functionality, are changing from time to time. In formal words, it is the *identities* of the executing threads that are changing. Changing the thread identities makes it difficult to associate a thread state with an particular executing thread. This in turn makes the resumption of execution easily go inconsistent. The *thread-state association preserving* issue would be further discussed in Section 5.5.2.

5.5.1 Reactive migration

As discussed before, support on *reactive migration* is a necessity in implementing thread-state migration. It is because all except one thread have to migrate upon receiving migration signal triggered from a thread. However, *fine-grained reactive migration*, which can trigger migration at any points of execution, is very expensive to be implemented. In that case, *migration checkpoints* have to be inserted everywhere in the source code, which is clearly unacceptable when the code size and performance of the resulting code are important consideration factors.

Therefore, we adopt the *coarse-grained reactive migration* scheme, which only triggers migration at some pre-defined points of execution. *Migration checkpoints* therefore just need to be inserted at some pre-defined points in the source code. If the pre-defined points are well selected, the reactive migration can still apparently respond to the migration signal simultaneously, at the same time effectively reduce the size of the resulting code and increase its performance.

In our discussion, we would encapsulate the migration checkpointing process into a method `checkMigrate()`. What this checkpointing method does is to just simply check and see if the *migration flag* on the shared object has been set or not. If it is set, the thread would undergo migration. The idea is illustrated in the following pseudo-code. It should be noted that the migration originator is responsible for setting the destination address of the migration in the corresponding attribute in the shared object.

```
if (sharedObj.checkMigrate())
   go (sharedObj.destinationAddr);
```

In our implementation, we adopt a simple approach in placing migration checkpoints: we assume migration checkpointing is to be carried out at *facet* granularity. In other words, we put these migration checkpoints at all of the entry points in facets, i.e. , the first statement of any facets would be the `checkMigrate` method call. It is a reasonable assumption since we regard the execution time of each facet should be small. Besides, some special migration checkpoints can also be placed in/after some synchronization blocks or inter-thread communication constructs. This would be discussed in later sections on code instrumentation for these constructs.

5.5.2 Preserving the association between threads and states

Problem description

Recall that in JavaGo (and in fact many other portable code instrumentation schemes on supporting transparent migration), state capturing is achieved through the use of the Java exception handling model: when execution state is to be captured and migrated, a special migration exception would be thrown. This exception would then propagate until the bottom of the stack, at the mean time constructing "stack frames" and saving execution states into a state object. Finally, *the current thread terminates*, and the "stack frames" are transferred to the remote site. There, a *new thread would be created* to resume the execution state. It should be noted that this scheme is perfectly correct in the single-threaded scenario: even if the identity of the executing thread changes, since no inter-thread communications are involved, the behaviour on the executed code would not depend on thread states, and therefore no problems would be introduced.

The situation is entirely different in the multi-threaded scenario. In that case, the code behaviour is tightly coupled with the thread states. The association between the thread identities and thread states become critical to the restoration of the execution status. For example, consider the following scenario: as we will see later in Section 5.7, a `waitSet` table, which is shared among all the threads and stores the state of the lock objects and the set of threads waiting on it, has to be used in our mechanism. If the thread identity changes upon migration, we cannot tell from the table whether a particular thread should wait on the lock object or not when the execution resumes on the destination site. Even worse, let us consider another scenario: a thread t_1 invoke the `join()` method upon another thread t_2 to wait for its termination. At the mean time, migration takes place, and both t_1 and t_2 terminates after the state-capturing stage. On arriving the destination site, the executions originally handled by t_1 and t_2 would be handed over to newly created threads t_3 and t_4 respectively. However, at that time, t_3 is still holding the identifier of t_2. Therefore, when t_3 resumes its execution and invoke `join()` method again on t_2, the execution would simply bypass the statement (because t_2 has already terminated), which makes the code execution erroneous.

Obviously, the above scenarios can be avoided if the association between the thread state and the identity of the executing thread can be preserved even after migration. That is, when the execution of the thread resumes, all the executions that involve inter-thread communication should not be applied directly to the thread identities that are acquired before migration. Rather, the executions should be applied to the thread identities that *owns the corresponding thread states* after the migration takes place. For example, in the second example above, t_3 should not invoke `join()` method again on t_2. Instead, it should invoke `join()` on the thread that takes over the execution of t_2, i.e. t_4.

The existence of the problem

The origin of the above problem comes from the fact that executions are handed over from one thread, say t_1 in the above example, to another thread, say t_3, upon migration. One may wonder, is it necessary for such scenario to happen in real situations? It may not necessarily happen, but it is the case under most situations. It is because threads in Java (in programmers' point of view) exhibit two properties:

- **Java threads are not serializable.** According to the description in the standard Java API, the `Thread` class in Java is not serializable. It follows that we cannot directly transfer a thread object to a remote site through object serialization. Programmers who wants to do so must extend the `Thread` class and implement the `Serializable` interface. But since there is no way to make executions of some "system" methods, e.g. the `main()` method for Java application or the *event-handling* methods, use these customized thread objects, the execution states captured in these system methods must be handled by newly created threads upon migration.

- **Java threads are not restartable.** It has been stated in the Java specification that: *Threads are not restartable, even after they terminate* [7]. However, most of the existing code instrumentation schemes terminate the execution of threads after capturing their execution states. In other words, even if these threads are successfully migrated to the destination host through serialization, they cannot be restarted. Their executions have to be handed over to other threads.

  One possible workaround towards this problem is to maintain a *thread pool*, in which the used thread are put into the *waiting state* rather than *terminating state*. However, implementing such a thread pool may make the logic of the code instrumentation scheme complicated and may therefore introduce more execution overhead. This scheme is therefore not analyzed at the time being and is left as future work.

Because of these two reasons, the mechanisms on handling the association between thread states and thread identities are still needed to be considered in the real world situations.

Our strategy towards the problem

The strategy to solve the above problems is simple. When a thread is created, a state identifier `stateID` would be assigned to the *execution state* of it. This `stateID` would also be transferred to the destination site when the execution state associated with it migrates. It essentially means that the `stateID` is bound to the *thread execution* throughout its whole execution lifetime, rather than depending on the *lifetime of the thread*. In other words, a new thread at the destination site that takes over the execution using a particular execution state would inherit the state's `stateID`.

By using this approach, the above problem is halfway solved: there is now a way to identify *thread execution*, which would not be changed even after migration. The wait table problem presented in the previous section can be solved: the `waitSet` table can now store the mapping between lock objects and the `stateID`s of the suspending thread executions. Since new threads at the destination site would inherit the `stateID` of the execution state it takes over, it now becomes possible to know from the table if that thread has been suspended before migration.

However, there is still a problem in addressing other thread objects after migration, just like the `join` example presented before. That is, there is no way for one execution to get the identity of the new thread executing another execution that it waits for. To tackle this problem, we maintain a *stateID to thread object* table in the shared object. When a thread starts its execution, it has to go through a *state binding* process. During the process, a `stateID` and the thread identity would be put into the table: if the execution is just a resumption of a migrated execution state, the `stateID` of the execution state would be used; otherwise, a newly created stateID would be assigned and used. Through this table, other threads can now easily address other threads by keeping the `stateID` of the thread's execution. Of course, similarly if the thread normally terminates, it has to go through a *state unbinding* process, in which the relevant entry in the table would be removed.

It should be noted that the above table in the shared object needs not be transferred to the destination site, since it is only meaningful for local use. It can therefore be declared as a *transient object*[3].

[3]A transient object in Java would not take part in the object serialization process. In other words, it would not be transferred to the remote destination.

5.6 Migrating thread state of synchronization blocks

5.6.1 The Java synchronization model

Synchronization is usually used to protect concurrently executing codes from exhibiting *race condition*[4], which can have devastating effects on the developed applications if happening. In Java, locking would be used as a mechanism of synchronization. For methods and blocks that have been bounded by the `synchronized` keyword, locking would *serialize* the concurrent execution inside the block.

There are some mechanics tied to the usage of the `synchronized` keyword in the Java synchronization model. These mechanics are critical to our code instrumentation design, since we have to follow exactly the semantics behind them. They are elaborated as follows:

In Java, every instance of the class `Object` and its subclass has a lock. Since `Class` instances in Java are also regarded as objects, they also have locks, but are usually used in `static synchronized` methods, which are described below.

There are two syntactic forms based on the Java's `synchronized` keyword: *block* and *method*. Block synchronization takes an argument of which object to lock. This allows any methods to lock on any objects. Block synchronization is always considered to be more fundamental than method synchronization, since any method synchronization can be transformed into an equivalent form using block synchronization. For example, a method declaration:

```
synchronized void f() { /* critical section */ }
```

can always be transformed into its equivalent form:

```
void f() {
   synchronized (this) {
      // ‘‘this’’ points to the current object
      /* critical section */
   }
}
```

Locking an object *does not* automatically protect access to the `static` fields of that object's class. Access to the static fields is instead protected via *static synchronization*. Static synchronization uses the lock possessed by the `Class` object associated with the class that the static member is declared in. It can be used by declaring the relevant method as `static synchronized`

[4]*Race condition* is a condition that is caused by the timing of events. It is usually associated with synchronization errors that provide a window of opportunity during which one thread/process can interfere with another.

(method synchronization), or by using the static lock for a class as an argument of a synchronized block (block synchronization). For instance, for a class `C` to apply block synchronization on static fields, it can write:

```
synchronized (C.class) {
   /* critical section with access to static fields */
}
```

Locking in Java obeys a built-in acquire-release protocol controlled only by the use of the `synchronized` keyword. As mentioned above, all lockings mechanisms can finally be transformed into a block structure. It follows that a lock is acquired on the entry to the `synchronized` block, and release on exit (either normal exit or through *exception*). When a lock is released, a randomly-selected thread that is block-waiting on the lock can acquire it.

Locks in Java are *reentrant*. It follows that a thread hitting `synchronized` block are allowed to go into the critical section not only when the lock is free, but also when the thread itself already possesses the lock. With this property, a `synchronized` method is now allowed to invoke another `synchronized` method on the same object (which also tends to lock the `this` object) without freezing up.

5.6.2 Difficulties in satisfying the synchronization model

There are three main difficulties that a code instrumentation scheme must overcome in order to satisfy the Java synchronization model:

1. **Problem on state capturing.** This problem exists on the code instrumentation schemes adopting the Java exception model to capture the execution state. Consider the following scenario: suppose in an execution, there is one thread t executing inside the synchronization block, while the others waiting on the lock of the same synchronization block. At this moment, t triggers the migration event. In the instrumentation scheme adopting exception-throwing approach, it implies that a migration exception would be thrown by t. By Java synchronization model, since t now releases the lock through exception, one of the threads waiting on the lock is able to go into the critical section and execute. However, in ordinary execution, since t has not finished its execution inside the critical section, other threads should not be allowed to go in. Therefore, the consistency of states inside the critical section is therefore violated. To prevent this problem from happening, we must ensure that no threads are

allowed to go into the critical section if the thread that was still accessing the critical section migrates.

2. **Problem on state resuming.** This is a typical problem that arises in migrating thread state: if a thread is inside the synchronization block before migration, how to ensure that it is the *first* thread inside the same synchronization block but not other threads? Since the lock for the synchronization block is acquired based on the first-come-first-serve manner, the problem implies that there should be a way to control the resuming threads, such that the thread that was inside the block is allowed to go in the block first.

3. **Problem on reentrant locks.** As mentioned in the previous section, Java locks are reentrant. It means that the resulting instrumentation scheme should not have any problems (e.g. deadlocks) even if the thread is acquiring a lock that is possessed by itself. Also, by reentrant locks, it means that the same lock can be acquired by the same thread more than once. It follows that the lock should not be allowed to be acquired by other threads if the lock is not released by its owner for the same number of times. The instrumentation scheme should also correctly deal with this scenario.

5.6.3 Our instrumentation scheme

As described before in Section 5.6.1, any Java `synchronized` methods, including both static and non-static methods, can be transformed into a more-generalized form, which essentially is a `synchronized` block in a non-`synchronized` method. Therefore, in the rest of our discussion, we would assume that the above transformation has been done and focus our discussion on the more general `synchronized` block.

Translating `synchronized` block

Translating a `synchronized` block is a bit difficult if the scheme of instruction point and stack state migration mechanisms in JavaGo are to be used together. It is because the contents inside a `synchronized` block are to be executed *atomically* [27]. In other words, executions are not allowed to jump suddenly from a point outside the critical section to a point inside, and the contents inside the `synchronized` block cannot be divided, thereby prohibiting the use of unfolding

Strategy 1 Splitting `synchronized` blocks

```
migratory void f(){
   /* Code before synchronized block */
   synchronized (obj){
      /* Critical section */
   }
   /* Code after synchronized block */
}
```

$\Downarrow$

```
migratory void f(){
   /* Code before synchronized block */
   SYNf();
   /* Code after synchronized block */
}

migratory void SYNf(){
   synchronized (obj){
      /* Critical section to be instrumented */
   }
}
```

technique and the top level jump technique in JavaGo[5].

In view of this, we instrument the code as follows: we add a new method (with name prefixed "SYN") for *each `synchronized` block* in the same class. The content of this new method is the `synchronized` block itself, while the invocation to this new method is put at the original place of the synchronization block. The original JavaGo instrumentation would then be applied to the contents inside the synchronization block of the new method. An example code instrumentation scheme is shown in strategy 1.

Through using this instrumentation approach, the code can be instrumented using JavaGo's scheme without violating the atomicity of the `synchronized` block.

Code instrumentation scheme on migrating locking states

Our code instrumentation scheme on migrating the locking states associated with the `synchronized` block is listed in strategy 2. It should be noted that we would use the variable `obj` to represent the argument inside the `synchronized` block, i.e. the owner of the lock being applied to the `synchronized` block.

[5]In fact, the code cannot even be compiled if there is a `synchronized` block spanning two `case` switches. We therefore cannot jump into a synchronized block from outside through using switch-case statements.

There are two important variables introduced in the instrumentation scheme: `syncLock` and `inSyncBlock`. They are used together to describe the current locking states of the executing thread.

- **syncLock.** `syncLock` is a serializable object that is used to simulate the behaviour of the JVM lock associated with object `obj`. Its value is set upon calling a method `getSyncLock` in `sharedObj` (line `03`), in which the corresponding lock object associated with `obj` would be returned through a table lookup inside the shared object. On the other hand, since this table, called the *lock table*, inside the shared object is shared among threads, if the value of `obj` being passed in is shared among threads, the lock object returned and stored in `syncLock` would also be shared among these threads, thereby achieving a simulated effect of *lock sharing*.

- **inSyncBlock.** `inSyncBlock` is a local variable that is owned by each individual executing thread. It is used to store whether the current executing thread has ever been in the `synchronized` block. Since each `synchronized` block is now put into a individual method after applying the instrumentation strategy 1, only one `inSyncBlock` is needed to be declared in the method and used dedicatedly for that `synchronized` block.

There is yet another variable `isResuming` appeared in the instrumented code. It is used as a flag to indicate whether the current execution is in a state resuming status. It can be set upon detecting whether the passed in state object is empty or not. In general, it is not dedicatedly used in the locking-state migration mechanism, but is instead used throughout the whole code instrumentation scheme, which would be seen in the later sections.

On the other hand, there are also several data structures and methods declared in the shared object that are used in the instrumentation scheme:

- **The lock table.** As mentioned above, the lock table is a data structure declared inside the shared object, and is shared among executing threads. It stores the `obj`-to-`syncLock` mapping, and plays an important role in sharing simulated locks among different threads.

- **The locking list.** It is another shared data structure that is declared inside the shared object. It is used to stores the list of `syncLock`s that are in a status of being locked. In other words, if a `syncLock` appears in the locking list, it implies that a thread is executing inside the `synchronized` block associated with `obj`, where `obj` is the object whose lock is being

Strategy 2 Instrumentation scheme for `synchronized` block

```
synchronized (obj) {
   ... /* Code to be instrumented */
}
```

$\Downarrow$

```
if ( !isResuming )
   inSyncBlock = false;
   syncLock = sharedObj.getSyncLock( syncObj );
}
if ( !inSyncBlock )
   synchronized ( syncLock ) {
      if ( sharedObj.isBlockLocked( syncLock ) ) {
         sharedObj.checkMigrate();
         syncLock.wait();
      }
   }
try {
   synchronized ( obj ) {
      sharedObj.checkMigrate();
      inSyncBlock = true;
      sharedObj.lockBlock( syncLock );
      ... /* Critical Section to be instrumented */
      inSyncBlock = false;
      sharedObj.unlockBlock( syncLock );
   }
} catch ( NotifyGone migGone ) {
   ... /* State saving codes */
   sharedObj.wakeAllThreads();
   throw migGone;
} catch ( Exception e ) { /* Some other exception e */
   inSyncBlock = false;
   sharedObj.unlockBlock( syncLock );
   throw e;
}
```

simulated by `syncLock`. Also, the `stateID` of the thread that acquires the `syncLock`, together with the number of times this lock has been *reentrantly* acquired, have also to be stored into the list. This is to identify the owner of the lock and enable the simulation of the behaviour of *reentrant locks*.

- **getSyncLock.** This method is used to find and return the *simulated* lock object associated with the passed in shared object `obj`, which essentially performs a table lookup on the lock table.

- **lockBlock.** This method is used to put the value of its argument `syncLock` onto the locking list, which in effect simulates that the associated lock of `obj` inside the `synchronized` block has been acquired. If the lock is reentrantly acquired, the count for re-acquirement of locks would be added correspondingly.

- **unlockBlock.** This method performs the reverse action of the above method, which removes the value of its argument `syncLock` from the locking list if the count for re-acquirement of locks is decreased to zero. In that case, it simulates the effect that the associated lock of `obj` inside the `synchronized` block has been released. In addition, it would also wake up all the threads that are block-waiting on the simulated lock `syncLock`, so that they can proceed their execution.

- **isBlockLocked.** This boolean method performs a checking on the locking list to see if the value of its argument `syncLock` is on the locking list. If not, the block is currently not locked and the method returns *false* to its caller. Otherwise, the identity of the lock owner is matched against the identity of the executing thread. This is *to simulate the effect of the reentrant lock*. If the identity matches, the block is not locked and the method again returns *false*. Otherwise, the block is locked.

- **wakeAllThreads.** This method will wake up all the threads that are block-waiting on the simulated lock `syncLock` for entering the `synchronized` block guarded by the `obj` object. It is used only when migration takes place, so that all threads can undergo the state saving stage. As one will see later, this method will also have some other uses when the code instrumentation schemes on `wait` and `notify/notifyAll` methods are discussed.

Next, we would split our discussion on the execution flow of our instrumented code into three parts: normal code execution, state capturing and state resuming.

Execution flow in normal situation

In normal executions, the value of the variable `isResuming` is being set to *false*. In that case, when the point of code execution goes into the instrumented method of the `synchronized` block, the values of `inSyncBlock` and `syncLock` would be initialized to their proper values (line `01` - line `04`). Line `05` is only useful during the state resuming stage, and is meaningless during normal code execution: the code from line `06` - line `11` would always be executed.

Line `06` to line `11` is also only meaningful during the state resuming stage. In normal execution, it will just try to simulate the working of a synchronization lock for *one locking phase*. By one locking phase, we mean that any threads that are being blocked by the lock would need to wait only until at most one unlocking operation is invoked by other threads. This is different from the normal behaviour of a synchronization lock: even if unlocking operation is invoked, the threads that are waiting on the block are not guaranteed to be freed from blocking, since only one of them can be chosen by the thread scheduler and continue the execution. The reason on doing so would be discussed later in the state resuming section.

The execution flow of this block of code is in fact trivial. It first performs a check to see if the simulated lock is acquired by other threads. If not, the execution can proceed. Otherwise, it has to be blocked from executing further by the invoking `wait` method on `syncLock`(at line `09`), until another thread calls the `unlockBlock` method on `syncLock` or the `wakeAllThreads` method when migration takes place. Both of these invocation would wake up all the threads waiting on `syncLock`.

After that, line `15` - line `19` performs the housekeeping work: once the executing thread is inside the `synchronized` block, it has to set its associate thread state `inSyncBlock` to be *true* to indicate that it is inside the `synchronized` block. After that, it puts the simulated lock `syncLock` onto the locking list by invoking the `lockBlock` method. The reverse action has to be done when it leaves the `synchronized` block. Notice that the same unlocking action has to be done when exception is thrown (line `26` - line `27`) in order to correctly simulate the behaviour of the JVM lock.

Execution flow when capturing state

One main problem arisen from multi-thread state capturing that must be addressed by any portable code instrumentation schemes which supports only *coarse-grained reactive migration* is the *mi-*

gration time mismatch problem[6]. In the instrumentation schemes, when the migration signal is received, there is no way to stop the execution of all running threads at a same particular instant and capture their thread states. In contrast, even if one thread finishes its state capturing, the other threads are still allowed to be executed until a pre-defined migration checkpoint is reached. In that case, problems suggested in Section 5.6.2 may happen. Therefore, any successful instrumentation scheme must ensure that the execution result of a multi-threaded program should still be consistent with that of a single-threaded one under such situation.

Our strategy on solving this problem is to prevent any threads to go into the critical section even after the thread owning the critical section finishes its state capturing. To do so, a `checkMigrate` statement is inserted immediately after the `synchronized` block (line `14`). In that case, if the thread that owned the critical section exit the block due to state capturing, the threads that try to enter the synchronized block thereafter would also trigger state capturing, and are therefore unable to execute the contents inside the critical section. Furthermore, since the values of simulated thread states of these "intruding" threads are not set to lock the block (i.e. they are not able to execute line `15` to line `16`), they will be considered as the threads that have never been inside the `synchronized` block on state resuming.

Similar principle applies to the other parts of the code: for the threads waiting on the simulated lock `syncLock` on line `09`, they will flow to line `14` after being waken by other threads, and migrate if a migration signal is triggered in between. On the other hand, to prevent the last thread from block-waiting at the `syncLock` forever while the other threads are ready to migrate[7], an extra `checkMigrate` invocation is inserted into line `08` so that it will migrate before it has a chance to wait at line `09`[8]

The rest of the state-capturing mechanism is then simple. Since the simulated thread states are maintained during normal code execution, and these states are kept inside serializable objects, they can be packed directly and transferred to the destination site through the standard Java serialization technique.

[6]In fact, this problem will always exist if the reactive migration is not so fine-grained as performing migration checkpointing per statement.

[7]It is possible to have such a scenario. Consider the following case: thread t_1 is in the `synchronized` block and has performed state capturing. At the mean time, thread t_2 is just about to execute line `06`. In that case, if line `08` is not present, thread t_2 will end up block-waiting at line `09` forever.

[8]It can be said in another way: if the thread has a chance to execute line `09`, it follows that there must be a thread to wake it up, since at least there is a thread inside the `synchronized` block that is not yet state-captured.

Execution flow when resuming state

During the state resuming stage, line `01` to line `11` have to be executed again. This time, the variable `isResuming` is being set to *true*, which in effect skips the variable initialization steps from line `01` to line `04`.

Line `05` to line `11` are useful in the state resuming stage. These lines of code now serves as an *execution sieve* that allows only the thread that owned the `synchronized` block before migration to go through and enter the `synchronized` block at line `13`. For all the other threads, they have to wait on the simulated lock object `syncLock` on line `09` until the lock-owner thread finishes execution and unlocks the block. This solves the resuming order problem described in Section 5.6.2. Also, since these lines of code are just used as an *execution sieve*, they need only to lock the other non-lock-owner threads for one locking phase. Once these threads are being waken by the owner threads, the responsibility on determining which of these threads to go into the `synchronized` block would go back to the standard Java synchronization construct at line `13`.

Since line `05` to line `11` are useful only when resuming state, it is tempting to modify the condition in line `05` to:

```
if ( isResuming && !inSyncBlock )
```

By doing so, the block would only be executed during the state resuming stage. However, it is incorrect. Consider the following scenario: a non-lock-owner thread, that has never visited a `synchronized` method, is resumed before the lock-owner thread of that `synchronized` method. After executing for a while, this thread enters the above `synchronized` method while the lock-owner thread is still not yet resumed. If the modified condition above is used, the non-lock-owner thread can now skip the execution from line `05` to line `11` and enters the `synchronized` block, which makes the state of the critical section inconsistent. Therefore, the condition `isResuming` must be removed from the condition so that every threads perform the block-locked checking at line `07`.

As a final note, the use of the local variable `isSyncBlock` is just to improve the execution speed. In fact, the same piece of information is also available in the locking list, where the owners information of the lock are stored. However, finding the locking list may involve many synchronization, which may effectively slow down the execution speed. A local variable is therefore used to indicate whether the thread has acquired the lock before migration.

5.7 Migrating thread state associated with `wait`, `notify` and `notifyAll` methods

5.7.1 The Java model on tackling these methods

In Java, just as the fact that every object has a lock associated with it, every object has a ***wait set*** that is manipulated by its methods `wait`, `notify`, `notifyAll` and an external method `Thread.interrupt`. This *wait set* for each object is internally maintained by the JVM. Each of this set holds the threads that is blocked by the `wait` method on the object until the corresponding notification methods are invoked. On the other hand, in Java, *locks* and *wait sets* work closely together. The methods `wait`, `notify` and `notifyAll` can be invoked only when the synchronization locks of the corresponding object are held by the current executing threads. In other words, these methods can be invoked only inside the synchronization block guarded by the objects that the methods are invoked on. Failing to do so may result in an exception being thrown.

The behaviour of the methods are briefly described as follows:

- **wait.** A `wait` method invocation would cause the current thread to be *blocked*[9] At the same time, the JVM would put the current thread into the internal *wait set* of the target object that the method is invoked on. After that, the synchronization lock of this target object held by this thread would be released (even if this lock is reentrantly held), but all other locks being held by the thread are retained. Upon later execution resumption upon notification, the lock state would be fully restore.

- **notify.** A `notify` method invocation would make an arbitrary selected thread, say t, be removed by the JVM from the internal *wait set* associated with the target object. There is no guarantee about which of the waiting thread would be selected if there are more than one thread. To resume execution, t must re-obtain the synchronization lock for the target object first. It follows that t is to be rescheduled by the JVM on entering the critical section. After t has successfully re-obtained its target lock, it can then resume its execution from the point where the corresponding `wait` was called.

- **notifyAll.** A `notifyAll` method invocation works in the same way as `notify` except that the above event occur for *all the threads* in the wait set for the target object. However,

[9]If the current thread has been interrupted, the blocking will be withdrawn immediately, and an exception would be thrown.

because these threads would compete for getting the synchronization lock upon resumption, the threads would continue execution one at a time.

In fact, there is another method that can affect the status of the wait set status: `Thread.interrupt`. It is invoked on a thread that is blocked on the `wait` method. It has the same effect as the `notify` method, except that after re-acquiring the lock, the execution throws an exception and the thread's interruption status is set to false. Since we think that the core mechanism on instrumenting this method is similar that in instrumenting the `notify`/`notifyAll` methods (except that some additional flags for interrupted state has to be maintained), the instrumentation of this method is left as our future work, and would not be discussed in our instrumentation schemes.

On the other hand, there are also *timed waits*. They have the same behaviour as normal `wait` method calls. The only exception is that if the time preset on these *timed waits* expires and no notifications are called upon the target object it waits on, the executing thread will resume execution. We have not designed the instrumentation scheme for this method also, since we think that it can easily be done if some states associated with the remaining time to wait on the target object are kept. The exploration on the instrumentation schemes associated with this kind of operation is left as future work also.

5.7.2 Main difficulties in instrumenting these methods

There are two main difficulties that a code instrumentation scheme must overcome in order to make the methods stated above work correctly even after migration.

Problem on finding the originator of notification

There are in general two sources of notification that can be applied on a thread suspending on the `wait` method during code execution: one is raised from the normal invocations of the methods `notify`/`notifyAll` from the other threads. The other is raised when migration occurs: upon migration, threads that are performing state-capturing are responsible to wake up those threads that are either *blocking on locks* or *suspending in the wait sets* by calling the `wakeAllThreads` method.

Owing to the fact that there are two sources of notification, the waiting thread must have a way to differentiate these two sources of notification, since they have different effects on future execution behaviour: While normal notifications would cause the thread to be removed from the wait set, migration notifications should not do so, since the thread should not have been notified under normal code execution.

One may think that this could be solved by just a simple instrumentation: inserting a `checkMigrate` invocation right after the `wait` method call. Then, if we encounter migration event just after notification, we classify it as a migration notification; otherwise, we classify it as normal notification. However, such a instrumentation scheme is incorrect. Consider the following scenario: a thread t_1 is waiting in the wait set of an object, and another thread t_2 notifies it. However, after the notification of t_2, it does not immediately release the lock associated with the object. t_1 therefore continues to wait until the lock is released. Eventually t_2 migrates without finished executing the rest of the `synchronized` block. Then, if the above scheme is adopted, t_1 would now think that it is being notified upon migration, rather than being notified from normal code execution. It may end up that t_1 will go into deadlock after migration, since it thinks that it has never been notified by any other threads.

Because of the fact that in Java we would not know the originator of the notification, it is not possible to use local variables to store the thread states associated with these methods, which is similar to the use of variable `inSyncBlock` in the instrumentation scheme for the `synchronized` block. The thread states must therefore be kept in some global data structures and be modified accordingly when the migration events occur. How it can be done would be described in our instrumentation scheme later.

The migration time mismatch problem

Similar to locking state migration, migrating states describing the wait set status also exhibits the *migration time mismatch problem* described in 5.6.3. There are two main problematic scenarios that may be arisen from the use of the above three inter-thread communication methods:

- There is a scenario that is similar to the one that occurs in the case for the `synchronized` block: a thread t_1 in execution notifies a thread t_2 that is blocking on the wait set of the target object, but t_1 migrates and releases the synchronization lock of the target object before it leaves the critical section. In that case, t_2 may have a chance to go into the critical section and change its internal status, thereby causing program inconsistency.

- Another possible scenario is exactly the opposite to the above scenario: a thread t_1 that is blocked on the wait set of the target object is notified upon migration and state-captured.

After that, another thread t_2 issues a notification on the target object. In normal situation, t_1 should have resumed its thread execution. However, the thread state captured in t_1 now still indicates that it is in the suspending state. Therefore, there is a problem in making t_1 resume its execution at the destination site without suspending on the wait set of the target object.

5.7.3 Our instrumentation scheme

To address the above problems for migrating states associated with the three inter-thread communication methods, we again adopted the behaviour simulation approach, in which the behaviour of the wait set is simulated before/after the corresponding method invocation. In the following subsections, we will look into the details on how our instrumentation scheme deal with the state migration problems for each of the above methods.

Core idea of our code instrumentation scheme

As discussed before in Section 5.7.2, since Java's language feature prohibits the identity of the notification originator from being known to the thread being notified, instead of using local variables to store the associated states, in our scheme, we used a globally shared *wait set table* placed in the *shared object* to store the *simulated* waiting set of the target object. In this table, the target object of the `wait`/`notify` method invoked on, together with the `stateID`s of the threads suspending on the waiting set of this object, are stored. Through maintaining the status of this wait set table by inserting corresponding table modification codes before/after the `wait`/`notify` methods, the thread state of the corresponding execution can be easily captured and migrated.

Through using the above mechanism, the problems stated in the previous section can be easily addressed. To solve the problem on finding the originator of notification, we can instrument the code in such a way that the execution will modify the entry corresponding to the target object in the wait set table upon receiving a notification call during ordinary code execution. On the contrary, on receiving a notification call just for migration, the execution will not modify any states of entries inside the table. This modification scheme is simple yet natural, since if we regard the wait set table represents a part of the thread state of the execution, we should not let the migration enabling code to alter the information inside it.

On the other hand, as described in Section 5.7.2, for the *migration time mismatch problem*, there are two scenarios that may make the program status inconsistent. The first scenario can be addressed by an approach similar to our way on solving the critical section entry control problem for the `synchronized` block: By simply adding a `checkMigrate` method invocation immediately after the `wait` method call, we can prevent it from further altering the status of the critical section.

The problem raised in the second scenario is much difficult to solve. To solve it, we must have a way to store the notifications generated by `notify` or `notifyAll` acted upon the target objects, so that when the captured states get resumed at the destination site, we can reproduce the effects of these notifications on the corresponding threads. In our instrumentation scheme, we use a so-called *notify token table*. In this table, the target object of the method invocation, together with its *notify token list*, are stored. When the `notify` method is invoked on a particular target object, a new notify token would be added into its associated *notify token list*. Similarly, when the `notifyAll` method is called, certain numbers of tokens would be added to the corresponding token list of the target object, so that the resulting length of the list is the same as the number of threads waiting on the target object's wait set. Through using this approach, a resuming thread would wait on the target object *if and only if* (1) it was waiting on the wait set before migration, and (2) there are no more notify tokens available for it on the notify token list for the corresponding target object.

This approach of using notify tokens, however, is easy to make the program inconsistent if the way on using it is not carefully designed. Consider the following scenario: two threads, t_1 and t_2, migrates to a site. Before migration, t_1 was suspended on the wait set associated to an object `obj`, and t_2 was on its normal way of execution. Suppose that the execution of thread t_2 gets resumed first, and it soon invokes the `notify` method on `obj`. However, since the execution of t_1 is not yet resumed at that time, a notify token is added in the corresponding token list. At the mean time, another thread t_3 comes in and waits on `obj`. If the notify token list is not handled properly, this notify token may be assigned to the late-coming thread t_3, which will never happen during normal code execution. The correct execution behaviour should be that t_1 gets notified instead of t_3.

A modified approach to tackle the problem in the above scenario is therefore designed. Each thread waiting on the (simulated) wait set of a target object would be associated with a flag called `notifyCandidate`. This flag is initialized to false when the thread is first put into the wait set. Now, whenever the methods `notify` or `notifyAll` is called, all the flags of the threads waiting on the wait set of the corresponding target object would be set to true, indicating that they are the legal candidates of being notify. By using this mechanism, the thread t_3 in the above example, with its `notifyCandidate` flag still set to false, would not get notified.

However, by this modified approach, the program may still exhibit some inconsistencies under some situations. For example, let us continue with the above scenario: under the modified scheme,

t_3 is unable to get notified and therefore wait on the target object's wait set. After some time, another thread t_4 comes in and also wait on the target object's wait set because it is still not one of the notify candidates. At this time, t_2 invokes `notify` method again on the target object. According to the modified scheme, both t_3 and t_4 becomes the notification candidates (i.e. with their `notifyCandidate` flags set to true). Since t_1 is not yet resumed, and t_3 and t_4 are the notification candidates, t_3 and t_4 get notified, which is not the correct program behaviour under normal program execution. (The correct program behaviour should be that t_1 gets notified and one of t_3, t_4 gets notified.)

From the above example, it can be seen that the origin of causing program inconsistency problem in the modified approach comes from the fact that: there is no scheme in the instrumentation method on revealing the order of invoking the `wait` and `notify` methods. In view of this, in our instrumentation scheme, we associate a *ticket* (`ticket`) attribute, which is essentially a number for ordering, to every thread entries of every objects appeared on the wait set table, and also every tokens of every notify lists that are in the notify token table. A global value of *current ticket* (`currTicket`) is also kept in the shared object.

The strategy on setting the values of the above attributes is as follows. `currTicket` is first initialized to a small value, say 0. When the method `wait` is invoked and the `stateID` of executing thread is to be put into the wait set of the corresponding target object, the value of `currTicket` would be copied to the `ticket` attribute of the same entry in the wait set of the target object. On the other hand, when the method `notify` or `notifyAll` is invoked, the `ticket` attribute of the notify tokens thereby produced will also be set to the value of `currTicket`, but this time, the value of `currTicket` is also incremented after that[10]. By maintaing the values of `ticket`s and `currTicket`, we can now make a rule to prevent the occurance of the above scenario: *a notify token for a target object `obj` can only be acquired by a thread waiting on the wait set of `obj` holding the ticket with value **smaller** than that of the notify token.* For example, for the above scenario, t_3 and t_4 can only compete to get the second notify token issued by t_2 under the above scheme, since their ticket values are too large to get the first notify token issued by t_2. By doing so, the ordering between the `wait` and `notify` invocations can be maintained.

[10]It should be noted that all these updating of ticket values are also protected by locks acquired by invoking the `wait` or `notify` methods, thereby ensuring no race conditions would occur.

Instrumentation scheme on migrating wait set states: `wait`

Our code instrumentation scheme on migrating the wait set states associated with the `wait` method invocation is listed in strategy 3. Same as that in describing the instrumentation scheme for the `synchronized` block, we would use the variable `obj` to represent the target object that invocations of `wait`, `notify` and `notifyAll` would operate on, i.e. the owner of the wait set (and synchronization lock) that these methods has to apply on. It should be noted that in the listed code instrumentation scheme, only the numbered lines corresponds to the instrumentation of the `wait` method. Other lines are only the instrumentation of the `synchronized` block. Also, these lines of code are kept atomic. That is, the *jump facilities* used in the code instrumentation of JavaGo cannot break the codes in between and allow execution to start at the middle of the lines.

On the other hand, it should be noted again that there is no local variables used to store the thread state of the program, since we cannot tell from local information whether a notification comes from normal program execution or migration event. All the thread states are therefore modified/maintained through the use of globally shared data structures in the shared object and their associate methods.

There are two new methods in the listed code. Their uses are explained as follows:

- **addToWaitSet.** Invoking this method would make the `stateID` of executing thread to be added in the corresponding entry for the target object `obj` in the simulated wait set table of the shared object. As mentioned before, the value of the `ticket` attribute in the entry would also be set to the value of the globally shared `currTicket`.

- **isInWaitSet.** This boolean method would determine whether the executing thread is in the wait set of the target object `obj`. It has a side effect: if the executing thread is in the wait set and the method finds that a notify token associated to `obj` is *available* to this thread[11], the method would remove the executing thread from the wait set and also the corresponding token from the notify token list of `obj`, indicating that it has been notified. If so, the method would return false, since the thread is no longer waiting on the wait set of `obj`.

The `wakeAllThreads` method here performs similar functionality as that described in the previous section. It would wake up all the threads that are waiting on the wait set of all objects in the wait set table. Since the invocation of this method will not add tokens to the token lists of

[11]Availability here also means that the value of `ticket` in the notify token has to be larger than that possessed by the thread.

Strategy 3 Instrumentation scheme for `wait` method invocation

```
synchronized (obj) {
   ... /* Code to be instrumented (A) */
   obj.wait();
   ... /* Code to be instrumented (B) */
}
```

$$\Downarrow$$

```
     ... /* Same as that in scheme for synchronized block */
     try {
        synchronized ( obj ) {
           sharedObj.checkMigrate();
           inSyncBlock = true;
           sharedObj.lockBlock( syncLock );
           ... /* Code to be instrumented (A) */
           isResuming = false;
           if ( !isResuming )
              sharedObj.addToWaitSet( obj );
           if ( !isResuming || sharedObj.isInWaitSet( obj ) ) {
              inSyncBlock = false;
              sharedObj.releaseReentrantLock( syncLock );
              obj.wait();
              sharedObj.checkMigrate();
              sharedObj.isInWaitSet( obj );
           }
           inSyncBlock = true;
           sharedObj.reAcqReentrantLock( syncLock );
           ... /* Code to be instrumented (B) */
           inSyncBlock = false;
           sharedObj.unlockBlock( syncLock );
        }
     } catch ( NotifyGone migGone ) {
        ... /* State saving codes */
        sharedObj.wakeAllThreads();
        throw migGone;
     }
     ... /* Same as that in scheme for synchronized block */
```

the target objects, it essentially differentiate notification raised from it and that raised from normal execution.

Execution flow of the above scheme

The execution flow of the instrumented code is in fact very simple. For normal code execution, before the execution invokes the `wait` method, it has to do some housekeeping work on the wait set and the lock of the target object. This includes adding the current thread to the simulated wait set (line `02`) and unlocking the simulated lock (line `04` to line `05`).

If the thread is waken by the migration signal at the time it is suspending on the wait set of the target object signal is triggered, since it is not normally notified by other threads and is still inside the wait set of the target object, the execution of line `07` to line `11` is therefore skipped. State capturing is then triggered by invoking `checkMigrate` method at line `13`.

If the thread is waken by the notification of another thread during ordinary execution, the associated `stateID` would be removed from the wait set by acquiring the notify token released upon invocation of the `isInWaitSet` method on line `07`. Furthermore, if the block is not locked by other threads (which have also got notify tokens), it can lock the simulated lock by setting appropriate values in the thread state variables. With these status properly set, the execution is now ready to migrate at any time.

On thread state resuming, same as the case for the `synchronized` block, all the threads that do not own the critical section have to wait at the simulated lock. Once they get a chance to go into the critical section, they will jump to line `01` (rather than line `06`) to continue execution. There, because the execution is on its way of resuming, the `stateID` of the executing thread *needs not* be put into the wait set of the target object again (line `01` to line `02`). Line `03` tests to see if the thread was in the wait set before migration or not. If not, it can continue its execution (to code part (B)). Otherwise, it has to go through the whole waiting steps again. This basically finishes the whole state resuming process.

Instrumentation scheme on migrating wait set states: `notify` and `notifyAll`

Our instrumentation scheme on migrating the wait set states associated to the `notify` or `notifyAll` method invocations are much simpler than that for the `wait` method. The instrumentation schemes on these methods are listed in strategy 4.

Strategy 4 Instrumentation scheme for `notify` and `notifyAll` method invocation

```
synchronized (obj) {
   ... /* Code to be instrumented (A) */
   obj.notify();
   ... /* Code to be instrumented (B) */
   obj.notifyAll();
   ... /* Code to be instrumented (C) */
}
```

⇓

```
     ... /* Same as that in scheme for synchronized block */
     try {
        synchronized ( obj ) {
           sharedObj.checkMigrate();
           inSyncBlock = true;
           sharedObj.lockBlock( syncLock );
           ... /* Code to be instrumented (A) */
           sharedObj.removeFromWaitSet( obj );
           obj.notify();
           ... /* Code to be instrumented (B) */
           sharedObj.removeAllFromWaitSet( obj );
           obj.notifyAll();
           ... /* Code to be instrumented (C) */
           inSyncBlock = false;
           sharedObj.unlockBlock( syncLock );
        }
     } catch ( NotifyGone migGone ) {
        ... /* State saving codes */
        sharedObj.wakeAllThreads();
        throw migGone;
     }
     ... /* Same as that in scheme for synchronized block */
```

It can be seen from the code listing that excluding the code instrumented for the synchronized block, only one method invocation (i.e. `removeFromWaitSet` or `removeAllFromWaitSet`) is needed to be added before the invocation of `notify` and notifyAll. What these methods do are simply adding corresponding numbers of notify tokens to the notify token lists of the target object `obj`. The notified threads would then get these tokens on resuming from the wait sets to maintain their thread states.

5.7.4 Several notes on our scheme for migrating wait set state

It should be noted that throughout the instrumentation scheme, all the tables / lists contains only serializable objects: wait table stores the `stateID` of the executing thread, the target object (which is expected to be serializable) and the value of ticket; whereas notify token table just stores the target object and the value of ticket. The whole thread state is therefore transferable to the remote site through object serialization technique in Java.

It may seem that the code instrumentation scheme is somehow simple. In fact, lots of efforts have been put to keep the contents inside the tables to be free from race conditions through using the synchronization constructs, at the same time ensuring that these constructs are organized in such a way that is free from *deadlock situations*.

Finally, one may already note that the value of variable `isResuming` critically determines the behaviour of thread resuming of wait set states. Incorrect setting of this variable's value may result in unexpected code execution: setting the value for too long would affect normal code execution, while setting it for too short may not be able to fully resume all the states. In view of this, in our design, we enforce that the value of `isResuming` is only used for the instrumented code for `wait`, and this instrumented code should be kept atomic. Therefore, by simply assigning the value of `isResuming` back to false after the instrumented code and after every invocations of the `migratory` methods, the problem is solved.

5.8 Migrating thread state associated with `join` method

In Java, `join` is a member method for the class `Thread`. Invoking the `join` method for a thread, say t_1, suspends the thread t_2 that calls the method, until t_1 completes its execution and dies. Since the method `join` simply checks to see if t_1 is executing or not, it will have no effect if t_1 have not started running or even created.

The main problem in migrating the thread state associated with `join` method is that: if t_1 dies just because migration takes place, and in fact it has not yet completed its execution, then it would be erroneous if t_2 continues to execute the codes after the `join` method. In that case, t_2 should migrate also and resume the joining status at the destination site.

With this in mind, we can formulate a simple instrumentation scheme. There are now two possibilities when t_2 finishes suspending on the `join` method: one is that t_1 finishes its normal execution; the other is that t_1 is being state captured and ready for migration. To differentiate between these two scenarios, we only need to check if the migration signal is triggered or not: if so, it is the latter case; otherwise, it is the former case. To tackle with the latter case, we just need to invoke the `join` method again on t_1 when t_2 resumes its execution at the destination site.

Strategy 5 Instrumentation scheme for `join` method invocation

```
   ... /* Codes before join() */
   t1.join();
   ... /* Codes after join() */
```

$\Downarrow$

```
     ... /* Codes before join() */
01   stateID = sharedObj.getStateID(t1);
02   while ( true ) {
03      sharedObj.getThread( stateID ).join();
04      if ( !sharedObj.isMigrating() )   break;
05      sharedObj.checkMigrate();
06      if ( !sharedObj.checkAlive( stateID ) )   break;
07   }
     ... /* Codes after join() */
```

However, another problem exists in adopting such simple approach. If t_2 resumes before t_1, further invocation of the `join` method on t_1 will have no effect, since t_1 has not started running yet. t_2 should therefore wait until t_1 resumes its execution before the `join` method is applied on it.

Therefore, we finally come up with our instrumentation scheme, as shown in strategy 5. Different from previous instrumentation schemes used on migration states of locks and wait sets, the above code is not kept atomic. In other words, the JavaGo instrumentation strategies (e.g. loop unfolding, jump facilities, etc.) would be further applied on the above codes to achieve instruction pointer state migration and stack state migration. We leave the code in a non-instrumented form for the sake of clarity. The readers can simply think that the execution migrates after executing line `05`.

As discussed before in Section 5.5.2, there is a problem in referencing other thread entities when migration takes place. It is because the underlying identities of threads have been changed upon migration. This problem is solved through the use of `stateID`, which is a unique and permanent identifier associated to the state of the thread. Therefore, in the above instrumentation scheme, one can see that we have used intensively the `stateID` and thread table lookup to reference a thread.

There are two new methods introduced in the above instrumented code. The first one is `isMigrating`, which simply reports if the migration signal is triggered or not. The second one is `checkAlive`. It helps the suspending thread to see if the target thread is still alive or not. “Alive” here refers to whether the target thread state has been migrated to the destination site. If not, the method returns false. Otherwise, the execution of this method would suspend until the target thread resumes its execution, and returns true to the caller afterwards.

Therefore, the execution flow of the above code segment can be explained as follows (we would use the notation same as above: t_1 is the target thread and t_2 is the suspending thread): at the source site, t_2 invokes `join` on t_1, same as that under ordinary execution (line 03). When t_2 is resumed from suspension (i.e. t_1 terminates), it checks to see if the resumption is due to execution completion of t_1 or a migration signal has been triggered. If it is the former case, t_2 will exit the `while` loop (line 04) and continue its ordinary execution of the codes after the `join` method. If it is the latter case, t_2 will migrate by triggering the state capturing process (line 05). At the destination site, t_2 will wait until t_1 resumes its execution (line 06) before it loops back to invoke the `join` method. Of course, if t_1 has terminated before migration and no state is transfered to the destination site, t_2 will also exit the loop (line 06).

5.9 Other instrumentations related to thread migration

Having discussed about the most complicated parts in migrating thread states, there is still one issue remaining: we have to decide where to place the `undock` block in a multi-threaded code. According to the definition given by JavaGo, `undock` block encloses the area of code to be migrated to the destination site. Codes outside the `undock` block are not supposed to be migrated. In other words, the stack frame created by the method where `undock` block is placed is the bottommost stack frame in the stack that is to be captured and migrated. In JavaGo's model, the placement of the `undock` block is decided by the programmer. However, under such a scheme, the programmer must be very clear that the execution of the remaining code outside the undock block would not have any effects on the migrated execution, since these corresponding stack frames would not be migrated and therefore re-execute / reinitialize the variables at the remote site.

Therefore, our approach is to place the `undock` block at a place where the whole execution of the thread can be captured. For example, in writing Java applications, we can enclose the contents of the whole `main` method using an `undock` block. Similarly, for multi-threaded programs, we can enclose the contents of the whole `run` method of a `Thread` or `Runnable` object to capture the whole thread execution. That is:

```
public void run(){
   undock{
      /* Original contents in the method */
   }
}
```

Using `undock` block for state migration has another property: the method using the `undock` block to migrate its content does not need to have its signature modified (because of the way JavaGo instrument the `undock` block). This property is beneficial to the methods whose signature cannot be modified, e.g. the `run` method in class `Thread` and `Runnable` objects, and the *call-back methods* in the event-driven programs. For these methods, the instrumentation scheme shown above can be used.

However, the case of the execution states in call-back methods should be specially handled. This is because in Java, all the call back methods are handled by one single event-polling thread in order to prevent deadlock situations on programs with Graphical User Interface (GUI). However, if the above code modification scheme is used, each executions of the call-back method would be captured in different thread states. Therefore, at the destination site, several different threads would be used in executing these thread states concurrently, which may end up in a deadlock situation. To deal with this situation, the corresponding thread states can be marked *exclusive* with respect to each other, so that the underlying MCS would not concurrently execute these threads. Although this approach is currently not implemented in our MCS, we think that this approach is feasible and is classified into our future work.

5.10 Limitations of our scheme

There are still limitations in the current implementation of our code instrumentation scheme. Some of them are essentially due to source-code level instrumentation, while some of them are not handled by all current portable code instrumentation scheme supporting execution migration. They are briefly listed as follows:

- Our instrumentation scheme still does not allow migration to take place in a class initializer, in an instance initializer, and in an object constructor. This is because it is difficult to migrate halfway created objects using the source code instrumentation scheme. However, as we have said before, since our facet programming model does not require facets to migrate in these areas, this problem is easily avoided.

- I/O (resource states, especially *fixed* resource states) and GUI states and functions (User Interface states) are the two data states that is not migratable under our current scheme. However, most of the other portable instrumentation schemes did not handle the migration of these states also. We believe that if our code instrumentation can be used with some proper *data* *state management* supports (described in Section 4.3) or under the shared memory environment where data state migration is not a problem, the whole code execution can be migrated freely even in the heterogeneous environment. For example, if the GUI states in Java can be serialized and described in a common data format, say XML, the execution related to it may be easily transfered to the remote site. In fact, a simple implementation on this strategy has been done by XwingML [19] (unluckily this project is discontinued at the time of writing).

- In our current implementation, we do not deal with executions with its behaviour depends on the variables of class `Thread`. It is because as we have discussed above, the underlying executing thread is always changing upon migration. If the behaviour of a code execution always depends on other threads (e.g. execution always needs to check if another thread is still alive or not), the behaviour of the thread that the execution depends on may not be the same as what the programmer desires (e.g. the thread dies because of migration). This may make the resulting execution wrong. However, even though we have not implemented and tested the claim, we believe that this problem can be addressed by instrumenting the corresponding code similar to that used in instrumenting the `join` method.
- As mentioned before, some inter-thread communication related methods, like `Thread.interrupt` and some timed wait methods, are not instrumented in the current implementation. However, we believe that instrumentation schemes similar to those for `wait` and `notify` methods can be applied to them.
- Also, at the current moment, no support of standard Java class `ThreadGroup` is given. However, again this should be easily implemented by maintaining the hierarchy of threads and thread groups in the shared object using the `stateID` together with suitable instrumentation in the code.

5.11 Future works on the portable support of transparent migration

There are still large rooms of improvement available on the instrumentation scheme for supporting portable and transparent migration in Java. On top of the future works stated in the previous section on improving the correctness and supporting features of the instrumentation schemes, some future works related to the performance of the instrumented code have to be done. These include:

- We would exploit the possibility of byte-code level instrumentation on supporting thread-state migration One of the reasons we adopt the source code instrumentation scheme is its simplicity and flexibility. However, it is expected that such source code instrumentation scheme may not yield good performance. Therefore, byte-code level instrumentation on supporting thread state migration is a good experiment candidate on achieving better performance.

- Since only one shared object is used in our instrumentation scheme, and all the threads have to synchronize on that object, this shared object obviously become the scalability bottleneck. In the future, we may try to find out schemes to distribute the thread states in such a way that locking would not happen frequently.

- As mentioned before, whenever migration takes place, the underlying executing threads will be changed once arrived at the destination site. We need to find a instrumentation method such that the underlying executing threads needed to be changed less frequently than the current scheme. A possible area to explore is on how to use the *thread pooling* technique intelligently.

- Finally, the current scheme of instrumenting the `wait`, `notify` and `notifyAll` methods always takes care of the most general situations. It may be too conservative and strict. Some scenarios in the example given above may not happen if some additional migration checkpoint are put at the appropriate places. In the future, we may need to find easy ways to make the scheme less strict, thereby removing some unnecessary state saving routines and improving the overall performance.

5.12 Summary

In this chapter, we have witnessed the fact that most current implementations that claims to portably support strong mobility actually does not cover the case for thread states. We used a scenario to analyze why this is so. Also, we presented the technical hurdles that have to be overcome in order to portably support thread state migration. These techniques hurdles include finding ways to support *reactive migration* and properly deal with the problem on the change of running thread identity upon migration. We then describe in details our source-code instrumentation approach on solving all these problems, together with the scheme on migrating the thread states associated with some main programming constructs: the `synchornized` block and the `wait`, `notify`, `notifyAll` and `join` methods. Finally, we have studied the limitations of our schemes and some possible future works that can be done related to the code instrumentation scheme. Being the first trial on implementing transparent migration of thread states, we hope that these strategies and their design ideas can contribute to the mobile agent community.

Chapter 6

The lightweight mobile code system

From the previous chapters, we know that supporting *strong mobility* and therefore *transparent migration* is critical in enhancing the flexibility and programmability of the mobile codes. In the last two chapters, we have gone through several portable approaches in supporting the transparent migration of mobile codes, or more specifically, mobile agents. These approaches include *source code level instrumentation*, *byte code level instrumentation* and *using the Java Platform Debugger Architecture*. We have also discussed our source code instrumentation scheme on supporting transparent migration of multi-threaded programs. Although all these approaches satisfy one of the critical properties of mobile agent/mobile code systems: strong mobility is either portably supported or simulated, they exhibit one common problem: the resulting size of the instrumented code (or code with debugging information inserted) has been unavoidably blown up. Also, some additional data states and execution states that are not needed in the ordinary execution would be introduced during the execution of the instrumented code. This growth in code and execution overheads reduces the benefits that are potentially achievable by the support of strong mobility. It makes the instrumented code even more unfavourable to be executed in the pervasive computing environment, in which most of the participating devices are small not only in terms of form factor, but also in terms of available resources such as memory and network bandwidth. To allow these instrumented mobile codes (with strong mobility support), and thus mobile agents, to be executed in these environment, there must be ways to trim down the resource usage of them. This is where our idea on designing a *lightweight mobile code system* comes in.

In this chapter, we would discuss the concepts on how it is possible to make the resulting mobile code system more lightweight. It may be misleading to say that we are to make the mobile code system more lightweight, since one may think that we want to make the platform hosting the mobile codes/mobile agents more lightweight. In fact, it is not our main concern. It is because the design of the mobile code platform can be made as lightweight as possible by removing some supporting yet non-core features from the platform (it is reasonable to assume that the resource usage of the platform is directly proportional to its supporting features). Furthermore, the resource usage of the platform is rather static and would not subject to big changes once it is installed on the target system, just like the case in operating system. Instead, our main concern is on *how the platform can support mechanisms that make the mobile code executions hosting on it become more lightweight*[1]. The resource usage of the mobile codes are more dynamic and unpredictable, and therefore are critical in reducing the overall weight of the system if some ways on reducing their resource usage can be proposed. Therefore, in the following sections, we will put our focus on how to make the executions hosting on MCS become more lightweight.

As discussed before in Section 4.2, a code execution can be described by three main *execution constituents*: code, data and execution states. To make the execution lightweight, we believe that we should find ways to make these execution constituents as lightweight as possible during migration and execution, at the same time retaining the correctness of the execution. In the following sections, we would look into these execution constituents one by one, and try to exploit the possible ways on reducing the memory or network bandwidth usage for the execution of mobile code.

6.1 Concepts related to code mobility

Before going into the details on how execution can be made lightweight in the mobile code environment, it is always beneficial to understand some basic concepts and design paradigms related to code mobility. After understanding the relative advantages and disadvantages of these design paradigms, it would be easier to formulate and find out which design paradigm(s) have to be used in order to achieve best effect with the constraints posed by the pervasive computing environment.

It should be noted that most of the concepts presented in the section follow the classification made by Fuggetta *et. el.* [6]. This classification gives the most comprehensive and logical justification to different concepts and interaction patterns related to *code mobility*.

[1]*Lightweight* here refers to the resource usage of the execution is low. Furthermore, resource here mainly refer to the *memory* and network bandwidth usage.

6.1.1 Terms and definitions

Three main types of ***components*** were identified in participating a normal execution: the *code component*, the *resource component* and the *computational component*. The ***code component*** holds the *know-how* of performing an execution. The ***resource component*** refers to the data or resources distributed over the network that are required by the execution. The ***computational components*** are the *active executing entities* that are capable to carry out a computation using the relevant *code* and *resource* component. In normal executions, the computational components refer to some typical execution units, like *threads*, *processes*, etc. The execution of these *computational components* are usually carried out on some execution platforms, which we commonly refer them as ***sites***. Only when all the *components* are available on the same site (or when the code and data components can be reached by references available on the same site even though their contents are on different sites), the execution can be actually started.

The *components* described above should not be confused with the term *execution constituents* described in the previous chapters. While *components* are the entities that actively participate in an execution, *execution constituents* describe the snapshot of it. For example, one can regard a snapshot of a *resource component* during execution as the *data state*, and a snapshot of how the *computational component* execute the code as the *execution state*.

On the other hand, no matter components host on the same site or different sites, they can affect the execution behaviour of one another through a series of events, called ***interactions***. When we design mobile applications or mobile agents using mobile codes, we may need to take into account many different aspects that may arise from different types of interaction, e.g. the locations of sites, the distribution of components among the sites, and the migration of components to different sites. Interactions among components on the same site is totally different from that among components on different sites. This difference can be expressed in terms of network latency, access to memory, partial failure and concurrency.

It is generally believed that interactions among the components residing on the same site are *less expensive* than those to the components residing on different sites. However, with this rule alone, we cannot design mobile code systems with optimized performance when such a wide spectrum of interaction possibilities can happen during executions. Therefore, we need to identify some reasonable *design paradigms* for distributed system exploiting code mobility, and design our system based on their relative merits and drawbacks in the target environment.

Design paradigms of the mobile codes are described based on the *interaction patterns* that define the relocation of and coordination among the components needed to perform a service. As far as code mobility is concerned, there are three common kinds of design paradigms: *Code on Demand*(COD), *Remote Evaluation*(REV), and *Mobile Agents*(MA). These paradigms are characterized by the *location of the various components* before and after the execution of the service, by the *computational component* which is responsible for execution of code, and by the *location* where the *computation of the service* actually takes place. We will briefly describe these three mobile-code design paradigms, together with the traditional non-mobile-code design paradigm, the Client-Server paradigm (CS), in the next section.

6.1.2 The design paradigms

In this section, the design paradigms for mobile codes and non-mobile code are described based on different interaction patterns among components. For easy elaboration, in describing the design paradigms, the following service request model would be assumed: a computational component A located at site S_A needs the result of a service. In the accomplishment of the service, another site S_B would be involved.

Client-Server paradigm (CS)

It is the most prevailing and common interaction design paradigm that is used in the Internet and *does not* involve any uses of mobile codes. In this paradigm, a static computational component B located at the server site S_B is offering a set of services using the code and resource component available also at S_B. The client-side computational component A at S_A requests the execution of a service by interacting with the server-side component B. As a response, B would perform the service by executing the corresponding code component and accessing the required resource component in S_B. It would then deliver the result of the service to A through an additional interaction. One typical use of this paradigm is the Remote Procedure Call (RPC) and object request broker in CORBA.

Remote evaluation (REV)

In the REV paradigm, the computation component A at site S_A has the code component that is needed by the service co-located with it, but it lacks the resource components required, which are located at the remote site S_B. In view of this, A sends the code component to the computation component B at S_B. With all the components available, execution of service can be performed there. Upon finishing execution, B would deliver the result back to A through an additional interaction.

Code on Demand (COD)

In the COD paradigm, the initial setting of the location of the components is exactly the opposite to that in the REV paradigm. In this paradigm, the computational component A already can access the resource components it needs for performing the service, which are co-located at S_A. But this time, the code component required is absent there. In this case, A interacts with a computational component B at S_B by requesting the code component that can be found there. B then interacts with A by delivering the corresponding code component. A can then perform the service required locally at S_A and get the result of service. This paradigm has been widely adopted in the current web applications (in the form of Java Applets, and possibly Java servlets, etc.).

Mobile Agent (MA)

In the MA paradigm, the initial setting of the location of the components is the same as that in the REV paradigm: the code component required by the service is co-located with the computation component A at site S_A, and the needed resource component is on the other hand located at site S_B. Rather than just moving the code component to S_B for execution like the case in REV paradigm, this time the computational component A, together with some intermediate execution states, would be *migrated* to S_B also. After A has been migrated there, it completes the rest of not-yet-executed code using the resource component available there. From this, one can see that MA allows the greatest level of flexibility.

Therefore, from the above descriptions, it can be seen that the MA paradigm is very much different from other interaction design paradigms. While COD and REV paradigms focus on the transfer of code components between computational components (and no execution states are involved), in MA paradigm the whole computation is moved to the remote site along with its execution states, the codes it needs and some resources that it needs for performing the service. In view of this, codes that want to follow the MA paradigm are usually implemented on strong mobility systems, where transfer of various states and codes is not a problem.

In conclusion, the mobile code paradigm enriches the possible interaction patterns between components. In most traditional paradigms where no mobile codes are involved (i.e. components cannot move to other locations during their lifetime), the types of interaction and therefore its communication quality (via local or remote interaction) cannot change. However, for mobile code paradigms, components can be freely moved. By changing their location of execution, the quality of interaction can be dynamically changed and reduce the interaction costs.

6.1.3 Paradigms and technologies

One should be very careful in distinguishing between *paradigms* and its related *technologies*. What we were talking in the previous section are the design paradigms for mobile codes, which describe how the mobile components would interact with one another to accomplish the task. It does not describe *how these interactions and migration of components are implemented*. This is where *technologies* come in and provide possible ways to make these "hypothetical" interactions become true in the reality.

Generally speaking, technologies are somewhat orthogonal with respect to paradigms: the paradigms should be able to be implemented by different technologies. In spite of this, using a suitable technology to implement a design adopting certain design paradigm may significantly reduce the implementation efforts of programmers. To illustrate this idea, let us consider the case in the MA paradigm. For a design using the MA paradigm, the concept of moving a computation component must be actualized and implemented by some technologies. However, if a platform implemented by a technology supporting only weak mobility is used, since there is no way to preserve the exact execution state upon migration, some appropriate data structures for saving and restoring the execution state of the computational component must be implemented. Also, some additional mechanisms have to be added in the existing code to capture and restore the execution states in these data structures. As shown in the previous chapter, the effort is not trivial. However, if a platform implemented by a technology supporting strong mobility is used, the computational component may be able to be directly mapped to the destination site by triggering only one single instruction. From this, it can be seen that choosing the correct technology is important in implementing different design paradigms.

It should be noticed that one should not mixed up REV paradigm with REV system although their names are the same. REV paradigm is a design paradigm that describes the interaction of components, while REV system is a one of the *technologies* that may be used in implementing an application designed using the REV paradigm. Similar case applies to the name "MA". When we talk about the MA *paradigm*, we focus on how various components would move between sites and interact. It is talking on the code mobility aspect. However, most of the current researches focus on MA *technologies*, which focus on not only code mobility issues, but also on other issues such as the intelligence of the agents, what common languages and protocol should the agents speaks, what security policies should be enforced, etc. Furthermore, the application developed using MA technologies do not necessarily follow strictly the MA paradigm. For example, Java Aglets is one of the most well-known mobile agent *technology*. There will be no residual execution on the source site once the agent migrates. It seems that it is similar to the "component migration" property of the MA paradigm. However, since it does not transfer execution states and the execution always starts at a pre-defined point, applications designed on it are only considered to be using a hybrid approach mixing the REV paradigm with the MA paradigm. In conclusion, *technologies* mainly deals with issues on the *implementation* side of mobile agent, rather than on the *theoretical* mobility model adopted by the mobile agents.

In fact, the design on our lightweight mobile agents (LMAs) used in our mobile code system is a *technology* that adopts a hybrid approach of two design paradigms: the Mobile Agent paradigm and the Code-On-Demand paradigm. More on this will be discussed in the next section.

6.2 On reducing the size of required code

Before moving on to discuss how it is possible to make the execution lightweight, let us summarize the main properties of the above three mobile code paradigms: the REV and MA paradigms allow code to be executed at the remote site, thereby allowing local interactions with the resource components located at the remote site and thus reducing the cost introduced by component interactions. COD paradigm, on the other hand, does not address the issue of remotely located resource components. Instead, it just enables the computational component to retrieve code components from the remote site *on demand*, and therefore provides a flexible way to extend dynamically the execution behaviour (including the *adaptiveness* of the execution) and the types of interaction they support.

As what we have discussed before, there are three execution constituents: *code*, *data states* and *execution states*. We can always make the execution lightweight by reducing the size of one of these constituents during migration or when they are used by the execution components during execution. The above summary on the three mobile code paradigms may shed some lights on finding ways to make the mobile code execution lightweight as far as the *code component* is concerned.

Current mobile agent technologies are gaining more and more attention to the computer industry by their special mode of execution. It is because with the artificial intelligence built into them, they can automatically and intelligently travel on the interconnection network, at the same time retrieving the information that their owner needs or doing works that their owner wants. They can also interact with other mobile agents and adaptively respond to the stimulation from the hosting environment using their built-in intelligence. More importantly, paradigms like REV, MA or the hybrid version of them are always used in designing these mobile agents. Together with the artificial intelligence equipped with them, it is generally believed that mobile agents are able to move themselves to where the required resource components locates and interact with them locally, thereby reducing the bandwidth and latency overheads of communication and improving the execution performance and the resource usage efficiency.

However, there is one major drawback in the *traditional* mobile agent technologies: they adopt purely the REV or the MA paradigm in their design. In that case, the mobile agents have to carry all the codes and resource states (and also execution states in MA paradigms) that are required for the execution along with them throughout their whole itinerary. Also, since the codes they carry are not supposed to be changed while traveling over their itinerary, these carried codes are supposed to be able to handle properly all possible scenarios that could happen on their travel. Even if not impossible, handling all the cases would significantly blow up the size of the code the agents carry, which further burdens the load of these mobile agents. Therefore, although the introduction of the mobile agent concept brings us the advantage of enabling mobile executions, since traditional mobile agents may not be small and flexible enough to meet different resource constraints in heterogeneous platforms, they are not suitable to be used for computation in pervasive computing environment.

Our approach on tackling this problem is to adopt a special type of mobile agents, our lightweight mobile agents (LMA). The core concept of the design of LMA is in fact very simple. Based on the observation that it is not advantageous to migrate all the *code components* to the remote site, we search for ways to achieve similar goals as the MA or REV paradigm (i.e. access the remote resource component by local interaction rather than remote interaction), but without transferring the corresponding code components. We achieve this by adopting a *hybrid approach* on combining the MA paradigm and the COD paradigm. In our approach, only the computational components, together with its associated execution states and member states, are needed to be migrated to the site with the required resource component. When it arrives the destination site, it would use the COD paradigm to load in the absent but required code components. After all the required components are available at the remote site, the execution can take place.

One may think that the above scheme only delays the time of loading in code components – since eventually all the codes would be executed, the total bandwidth consumption of bringing in the code components would be the same. This seems to be true at first glance, but not necessary so if one thinks carefully. There are several ways in which code components in an execution can be made lightweight by this COD approach throughout the agents' itinerary:

- The agent may not execute *all* the required codes on only *one* site. Instead, it may execute a portion of the codes on one site, while leaving the others to be executed on the other sites. In that sense, bringing *all* the required codes every time when the agent migrates to a new site may be just a waste of bandwidth and memory, and probably time. If COD is used, only the portion of code that is needed during execution is brought in, thereby reducing the overall bandwidth and memory consumption throughout the whole itinerary.

- Similarly, the agent may not execute *all* branches of code components that an application uses. For example, the design of a mobile agent may involve some decision-making routines to select the appropriate strategies (which is composed by *code components*) to be used under particular scenarios. In that sense, only a portion, but not all, of the strategies are needed to be used. Again, in this case, bringing all the codes along with the agent is a waste of resources, and the use of COD paradigm address this problem nicely by bringing *only* in the required codes on demand.

- Even if the code execution requires that *all* the codes should be loaded on *every sites* along the itinerary of the mobile agent, it may still be unwise to bring along with it all the required code components. The required code components *may be already cached in the sites nearby or even at the destination site*. It would be faster and reduce the overall network traffic if the code components are to be fetched from these site rather than brought from the source site.

The above three points are the advantages of the use of COD paradigm in conjunction with the *modified* MA paradigm (where no code components are transfered during migration) in general situations. In our Sparkle architecture, there is one additional advantage in using this hybrid approach. In our architecture, the applications are designed based on functionality dependencies and compositions. Facets, which are one of the entities in our system and are essentially a specialization of code components, would obey the contracts set by these functionalities and implement their associate functions. By this, the application can still achieve the same goal (i.e. perform the same overall functionality) even if the underlying facets configuration are continuously changing.

On the other hand, different facets have different implementations. It follows that they would have different resource requirement and therefore different performance on accomplishing the functionality. Therefore, if the traditional approach of bringing all the code component along with the agent is used, the destination site may not have enough resources to execute or even store the code component it brings in. On the contrary, if the COD approach is used, since no codes would be brought to the destination site, different facets implementing the same functionality can be used. Some flexible and adaptive schemes can therefore be applied. In particular, for the above scenario, we can choose to load in another facet consuming less resource that is affordable by the environment of the destination site. Such a scheme on using the COD paradigm is lightweight and adaptive to the small devices in the pervasive computing environment. Interest readers on functionality/code adaptation are encouraged to our team members' thesis [20][39], in which a more detailed discussion and thorough analysis on this topic would be presented.

There is even one more advantage that is irrelevant to the mobility issues, but instead related to security issues. Since now facets would only be loaded on demand by LMA, irrelevant intermediate hosts will not be able to spy on the important algorithms or strategies on decision functionalities/facets carried by the agent. Also, they may not be able to understand the meaning of the states that the agent carries. In this case, the threat that the mobile agent suffers from malicious tampering can be greatly reduced.

There is, however, one main disadvantage to such a scheme. Because the required code components are brought in on demand, some additional time are needed to be spent on searching and loading the code components. But luckily, mobile agent technologies are in general using in applications that adopts *asynchronous interaction strategies*. In that case, the users of the application would not have to wait for the results of the execution before proceeding. They just need to send out messages at regular time interval to see if the results are available. Therefore, in this kind of scenario, time is not a major concern. Also, the loading and searching performance of code components can be improved through using the techniques like caching on the local site and pre-fetching on the proxies. This problem can therefore be easily addressed.

6.3 On reducing the size of data states

There are no general methods for the mobile code systems to reduce the size of data states, in particular member states, during code execution and migration. Common methods in reducing the size of data like *data splitting/streaming*, *data compression* and *data transcoding* cannot be applied in this scenario. It is mainly because we do not know the way in which the data is accessed (sequential or random access: data splitting/streaming is useless in making execution lightweight if the data is randomly accessed during execution), and there is no general way to reveal the semantic meanings carried by a randomly selected piece of data (transcoding therefore cannot be used in such scenario, since it does not know which portion of data can be dropped while retaining the information it carries). Data compression can help reducing the bandwidth requirement during migration of data components, but since all the data components has to be decompressed while executing at the destination site, it still does not help in reducing the memory requirement during execution. Therefore, if the data component is too large to fit on a small device, execution cannot be carried on it.

In view of this, programmers of the application need to be aware of the size of the data component to be transfered during migration. Since they know all the details about the data component, e.g. the way the code component will access on it, the semantics behind it, etc., they are responsible to provide ways to reduce the resulting data size upon migration (e.g. using the ways mentioned in the previous paragraph, or as the case in Java, marking the member states that are not needed to be migrated as `transient`.). With this, when the execution needs to migrate to a site with less resources, the underlying execution platform can use the routines provided by the programmer to reduce the size of the data components accordingly.

It should be noted that the above scheme are applied only when the involving data states *must be migrated* (e.g. according to the definition of mobile agents, the *member states* of the mobile agents must be migrated in order not to leave any residue states on the source site). For some other data states that are not supposed to be migrated during execution, such as the resource states or some private data states, they should be handled by the corresponding *data state management schemes* as presented in Section 4.3. Usually, these mechanisms are lightweight, since they just involve the movement of resource references.

6.4 On reducing the size of execution state

6.4.1 The origination of the idea

At first glance, the nature of execution state seems to be the same as that of data state. There is no general way to reveal the semantic meaning represented by the captured execution states, or more specifically, the *stack states*, since other types of execution states such as *instruction pointer states* and *thread states* are usually kept by the underlying system, and thus their semantic meaning should be known. Therefore, size-reducing techniques, such as transcoding, cannot be applied to it. However, different from data states that may be randomly accessed by the code, execution states are in general accessed sequentially following a pre-defined execution order: First-In-Last-Out on the execution stack. This property of execution state may shed some light on making the execution state lightweight during execution migration and resumption on small devices.

As what we have mentioned when we discussed about the instrumentation schemes in the previous chapter, the stack state and its associate instruction pointer state of an execution are modeled in a form of *stack frame list*. Each element in the list is a *stack frame*, which describes the current state of a invoked method during execution, including what the execution has done and where in the code the execution has been executed up to. These stack frames are stored in the list *according to the invocation sequence of the methods they are representing*. That is, the *head* of the list stores the stack frame representing the method that is *first invoked* when the execution starts, and the *tail* of the list stores the stack frame representing the method that is *most recently invoked* before migration. In another point of view, the head of the list corresponds to the bottom frame of the execution stack, whereas the tail of the list corresponds to the top frame of the stack. In most of the instrumentation schemes supporting transparent migration on weak mobility systems (like Java), in order to resume the execution state of an execution, the execution stack has to be rebuilt from the stack frame list that is transfered to the remote site during migration. The stack is rebuilt by traversing the list starting from the head frame, during which the local values are restored and the executed instructions are skipped using the stack states and instruction pointer states stored in the corresponding stack frame.

One possible point on making the execution state lightweight is based on the observation that: although upon execution resumption the execution stack has to be rebuilt according to the sequence of the stack frame in the stack frame list, in most cases, the restored information of a stack frame on the execution stack are not used until the stack frame on top of it (i.e. the stack frame after on the stack frame list) pops out, which happens when its associated method finishes its execution and returns. In other words, if we have a way to get rid of these "temporarily useless" execution states and migrate/restore the information associated with these stack frames to the execution stack *only when* they are required during execution, it is possible to make the execution more lightweight. This idea forms the basis of our strategy on reducing the size of the execution states[2].

6.4.2 Our strategy and its potential benefits

Based on the above observation, a simple framework of our strategy can therefore be formed. The basic idea of our strategy is illustrated as follows:

A figure illustrating the strategy may be needed here...

1. Given a list of stack frames, we chop the list into several different parts, with each part corresponds to an *state segment* that is to be transfered to the remote site individually.

2. Next, we migrate and execute the last state segment first.

3. Just before the last state segment finishes executing (i.e. hitting the return point of a method), its execution state is again captured and stored in a stack frame S.

4. Upon detecting this action, the underlying system or data structure would trigger a event (e.g. by sending a message) to the site where the second last segment resides, and get this segment there.

5. After getting the segment, the newly captured stack frame S appends itself to the end of this second last segment.

6. The execution of the second last segment can then start.

7. After finishing the execution, the above process would be repeated starting from step 3 until all the available execution segments finish their execution.

In simple words, the above strategy is in fact a data-splitting technique that aims to reduce memory and network bandwidth usage. By "streaming" in and execute the execution state segments one by one, this *state-on-demand* scheme can potentially make the execution lightweight with the similar arguments as that for *Code-on-demand* scheme: based on the discussion above, the agent execution may not depend the information recovered from the stack frames at the bottom

[2]Since thread state is not our main focus of discussion in this chapter, *execution state* in this chapter mostly refer to *stack state* and *instruction pointer state*, unless otherwise specified.

of the stack. Therefore, bringing *all* the execution states every time when the agent migrates to a new site in that sense may be just wasting the network bandwidth, memory and probably time. However, if the *state-on-demand* scheme is adopted, only the required state segment is brought in and executed, thereby reducing the overall bandwidth and memory consumption if the agent hops frequently throughout its whole itinerary.

6.4.3 Our implementation

In order to support the above *state segmentation* technique, data structure of the *simulated* stack frame and the underlying system should be modified accordingly. Luckily, since the definition of the `StackFrame` data structure is well-suited for our purpose, we do not need to make large changes on the instrumentation scheme to support this additional feature.

Briefly speaking, we need to make three main modifications to the data structure of stack frames, the execution code and the underlying system[3] respectively: for the data structure of stack frame, we need to add in a variable `isBottomOfSegment`. This variable would be set by the underlying system and indicates whether the current stack frame is the bottommost stack frame in the stack segment after splitting. It is useful for the state capturing invocation before its associate method finish executing. For the execution code, we need to instrument it in such a way that for every `migratory` method in the code (i.e. methods that may trigger migration during execution), we need to add the following statement before every possible returning points of the method, including those inside the exception handler:

```
if (isBottomOfSegment) throw new ReachBottomException();
```

The `ReachBottomException` above serve the same purpose as `NotifyGone`, in which it helps to capture the execution state of the current method. However, upon receiving this exception, the underlying system would respond differently from that when receiving the `NotifyGone` exception.

A figure illustrating the flow of code may be needed here...

The execution flow of an execution under the resulting modification scheme is therefore as follows: at the source site, the system would chop the stack frame list into state segments. For the stack frame at the head of each of these segments, their variables `isBottomOfSegment` would

[3]In our architecture, it is the *container* that should provide such support, since it manages the execution stack. For further details, please refer to Chapter ch:implementation.

be set to *true*. The last state segment is then migrated to the destination site and executed using strategy described in the previous section. When the execution reaches a point in the bottommost stack frame of the state segment that is just about to return, it hits the instrumented code listed above and throw an `ReachBottomException` exception. This exception would then trigger the underlying system to capture the execution state, find the next state segment, append the captured stack frame to the segment and execute the resulting execution state. This whole process repeats until the underlying system cannot find any pending state segments.

6.4.4 Implementation hurdles and limitations of the scheme

Although the strategy presented above seems to be correct and able to make the resulting execution and migration of execution states lightweight, it turns out that there are some additional implementation hurdles and performance overheads that is primarily introduced by a major language feature in Java: the Java serialization mechanism.

The Java serialization mechanism basically enables the objects in Java to be represented in the form of byte stream, and therefore can be *serially* written to different I/O streaming media like files, sockets, etc. easily while preserving the relationships between them. The working principle of it is in fact very simple: when we wants to *serialize* the specified object, it will save all the values inside the variables of the object, and then recursively traverse all the references to the other object and save them all in a similar manner. On the other hand, when a serialized object is *deserialized*, the original values stored inside the corresponding variable would be first restored, and then all the object's references to other objects will be traversed recursively to deserialize (and create) all objects that are reachable from it in a similar manner. By this way, the relationships between objects can be maintained even when the serialized object is exported to another site.

The Java serialization mechanism has no problems on deserializing the objects and the interrelationships between them if all the objects involved are deserialized *simultaneously*. However, the case becomes complicated when the objects involved are separately deserialized in different sessions. Consider the following scenario: there are two serializable objects O_1 and O_2, and both of them are holding the same reference pointing to another serializable object O_3. If O_1 and O_2 are transfered to the remote site by serialization but are deserialized separately (e.g. O_1 and O_2 are transfered through two individual `writeObject()` call), because of the Java serialization behaviour described in the previous paragraph, the references that O_1 and O_2 hold will no longer be the same. They will now be pointed to two different objects containing the same content as O_3.

With this property of object serialization in Java, the strategy we mentioned in the previous section is no longer trivial to be implemented. Consider the following scenario: S_1 is the stack frame derived from method M_1, S_2 is the stack frame derived from method M_2, and M_1 calls M_2 during execution (i.e. S_1 is below S_2 on the execution stack). Also, both M_1 and M_2 possess a local variable whose values are both a reference pointing to the same object O. In other words, when state capturing takes place, S_1 and S_2 should have a common reference pointing to object O. Therefore, if S_1 and S_2 are split and transfered in different serialization session due to the state segmentation mechanism, it ends up that S_1 and S_2 point to different objects during execution resumption, thereby making the execution incorrect. It should be noted that for the case above, O can be an object that is globally shared between M_1 and M_2, or a local object in M_1 whose reference is passed to M_2 through parameter passing.

Up to now, there is no clean solution to solve this *reference-preserving* problem. There are some ways to solve the problems, for example, *overriding the default serialization mechanism by using the `Externalizable` interface*. However, most of these strategies require the programmers to follow some rules on designing the classes of the method parameters (e.g. for the above strategy, it requires every method parameters to implement the `Externalizable` interface). Clearly, this approach limits the programmability of our model.

To address the above implementation problem, we have to rely on the definition and design of *facet*. According to the facet's definition, a shared reference is only possible to be acquired through its interface, since the facet programmer should not assume that the facet knows the existence of a globally shared reference (or otherwise it destroys the anonymity of the facet design). Therefore, if we place a restriction that splitting of frames can occur only when facet-calling takes place, the case in the above example that "O can be an object that is globally shared between M_1 and M_2" would not happen. On the other hand, facet has the following execution interface due to its generic design:

```
public Object[] execute(Object[] args);
```

If we instrument the code in such a way that the reference to O (which is one of the entry inside the array `args`) in M_2 is copied to the corresponding entry in `args` every time when the state is resumed, then even if the stack frames S_1 and S_2 are transfered in two different serialization sessions, M_1 can still get the object reference of O in M_2 through the `args` array. This basically solves the problem.

In fact, the above serialization property can induce to even worse situation. For example, two threads share a common object. In the current implementation, we transfer the execution states of all the threads at the same time, and therefore there is no error in the reference setting upon migration in the instrumentation scheme. However, in the case where execution states have to be split and transfered in several sessions, the reference problem described above may occur. We believe that the reference shared between threads is difficult, if not impossible, to be recovered once the sharing relationship is lost. Also, the states of the lock and the waiting set should be more carefully maintained, especially in the case of the re-entrant locks, where an additional *entrance count* should be kept in order to prevent other threads from coming in to the critical section when these locks are not yet fully released. Therefore, in the current implementation, we do not implement state segmentation for multi-threaded programs. In the future, we may exploit the possibility on designing some construct to easily specify the recovery semantics of the shared references.

As a remark, the behaviour of Java serialization reduces the amount of bandwidth saved by adopting the segmentation scheme. It is because the shared object O is transfered twice (once in transferring S_1, the other in transferring S_2) even though only one copy is used. Although the resulting scheme can still be named "lightweight" since it consumes less memory on maintaining execution states during executions, in some extreme case, using the segmentation scheme may even waste more network bandwidth than not using it.

6.4.5 Final remarks on the scheme

Excluding the negative effects and overheads introduced from the Java implementation of the scheme, the above state segmentation scheme should have reasonable effects on saving memory and bandwidth consumptions. However, same as all on-demand schemes, the above scheme exhibits the problems on *runtime performance degradation*. It is because additional time are needed to search for and bring in the corresponding execution state segments. Even if so, there are still rooms for optimization on the performance of the segmentation scheme:

- **Optimization on the way of segmentation.** The way in which the segmentation is done is critical to the performance of the resulting scheme. Too fine-grained a segmentation would make the stack frame retrieval routine be triggered frequently, which in turn increases the network traffic and slow down the performance. On the other hand, too coarse-grained a segmentation would make the scheme to migrate together the frames that are very unlikely to be used in the future (e.g. the bottommost stack frame). Therefore, some analysis may have to be conducted (in future) in order to strike the balance between the two and find the optimal segmentation scheme. At the current moment, we believe that some non-uniform segmentation scheme (e.g. large segmentations for the stack frames on the top of the stack and small segmentations for the stack frames at the bottom of the stack) can achieve nearly optimal performance, because the time of a stack frame being unused (i.e. more likely to be unused in the future) is expected to increase exponentially with decreasing height in position of the stack frame in the execution stack[4]. However, this optimization is possible only when the destination site has enough resources to hold even the largest segment resulting from the above segmentation scheme. If the destination site has only very little resources, maybe we still need to stick to very fine-grained segmentation and the corresponding performance unavoidably degrades in that case.

- **Optimization on state segment allocation.** Same as the case for the Code-On-Demand scheme, if the state segment can be placed on the site where the agent will migrate to and use the information in the segment before the agent actually migrates there, some times on searching and bringing in the state segment can be saved. However, this requires the the aid of programmer to tell or predict the possibility where the execution would go before the distribution of stack frames.

The above two optimization techniques can be further explored in the future. On the other hand, since some execution state segments are still left on the source site, this behaviour may contradict the definition of mobile agent technologies, in which no residue states should be left on the source site. This problem, however, can be again solved by distributing all the state segments on the site other than the source site and maintaining their locations accordingly after distribution. The algorithms similar to those *mobile agent tracking strategies* should also be applicable in this case.

6.5 Summary

In this chapter, we have covered two on-demand strategies in making the size of the resulting mobile codes lightweight. We proposed to use the *Code On Demand* code mobility paradigm in conjunction with the *Mobile Agent* paradigm to reduce both the size and the movement of codes over the network, thereby achieving an overall effect of bandwidth reduction. Furthermore, we have presented

[4]The time is expected to be increased almost *exponentially* with the height of stack frame in the stack simply because the code execution is in fact a in-order execution tree walk.

a similar on-demand scheme in migrating execution states (in particular, stack states), which also aims to reduce the overall number of stack frames transfered and thus the bandwidth usage. Besides, we have also gone through the technical problems in implementing these features in Java, including the limitations being placed by the Java serialization mechanism, and our workarounds to deal with it. Finally, we have proposed some potential optimizations in the state-on-demand mechanisms that can be done to further reduce the bandwidth consumptions. We would briefly evalute the effect of these optimzation strategies in Chapter 8.

Chapter 7

System design and implementation

In the previous two chapters, we have covered the strategies and mechanics on implementing the two core ingredients in mobile code systems. In chapter 5, we discussed how portable support of strong mobility on weak mobile code systems can be done on even multi-threaded codes. In addition, in chapter 6, how *lightweight migration and execution* can be achieved in these mobile code systems through the on-demand retrieval mechanisms of code and execution states is also analyzed. In this section, we are going to see how we incorporate these two core technologies into our mobile code system design so as to exploit the *computation flexibility* provided by the mobile codes and retain the *lightweight property* of the system. In doing so, we have made our mobile code system perfectly suitable for pervasive computing environment.

In this chapter, we shall first describe the core entities in our architecture and their corresponding functionalities and responsibilities. After that, we shall briefly describe how these entities would interact with each other to provide flexibility and adaptability to agent execution. Finally, we would discuss how a facet programmer can modify their facets to support transparent migration.

7.1 Main entities in our mobile code system

There are three core entities in our mobility system. They are the *Lightweight Mobile Code System* (LMCS), the *Lightweight Mobile Agent* (LMA), and the *Container* abstraction. They are operating at different architectural levels, and each of them have their individual responsibilities, which combine together to support various modes of mobility patterns, thereby providing flexibility to the agent/facet programmers. We are going to look into the details of them in this section.

7.1.1 Lightweight Mobile Code System (LMCS)

The *Lightweight Mobile Code System* (LMCS) is one of the entities resided in the client side system to support flexible computation by enabling data and code mobility. It is adapted from mobile agent systems (MASs) and resembles the concept of *site* described in the previous chapter. Also, as mentioned before, the reason why we use the name *Mobile Code System* rather than *Mobile Agent System* is that the executions hosted on the system do not exhibit only the property of the MA paradigm, but also the property of COD paradigm (in order to make execution flexible and lightweight). In spite of this, most of the designs in our LMCS still follow those in MASs, and *mobile agents* still play a critical role in our system (i.e. it is still reasonable to compare our LMCS with traditional MAS, since they are still similar in terms of their supporting functions).

Acting as an underlying layer, the LMCS is responsible for handling the activities of the mobile agents hosting on it. Briefly speaking, these include the creation, migration, and probably removal[1] of the mobile agents hosting on it. In addition, some agent controlling strategies, e.g. *agent location tracking* and *agent termination*, (which are already mentioned in Chapter 2) should also be supported. Through supporting these strategies, some mobile agent specific features can be easily implemented, e.g. agent retraction (which can be done by co-ordinating with the location tracking mechanism: by locating the agent using the mechanism, we can send a retraction signal to the located site to retract the agent). To support all these agent-specific features, communications and coordinations between LMCSs are required to be well established and maintained. On top of that, LMCS should also be in close contact with the central manager of the client system in order to get some useful information for aiding it to exercise the agent-governing policies (e.g. it can get the current resource usage information from the central manager and determine if any agents hosting on it should be migrated to serve the load-balancing purpose, or it can get the cache status to see if the code for the agent is already available on the local site).

There are in general two kinds of components that can be hosted on the LMCS: the *Lightweight Mobile Agent* (LMA) and the *container*. These two entities cooperate to function like a traditional mobile agent. They are decoupled from a typical agent because we want to separate the basic agent functionalities (e.g. migration and its security issues, interaction between agents, etc.) from the actual tasks being done by the mobile agents. Although the agent task may be tightly coupled with the migration semantics of the agent, we want to keep it separate so as to increase the flexibility and the maintainability of the agent programming model. We would look into the details of these two

[1]Removal of agents only apply when the agents violate the policy of the MAS/MCS.

entities briefly in the next two subsections.

7.1.2 Lightweight Mobile Agent (LMA)

The Lightweight Mobile Agents (LMAs) are the mobile agents that roam over LMCSs. It is different from the existing mobile agents in that they do not contain the tasks that they want to execute on the remote host. Rather, the tasks are carried by the *containers*, which would be dynamically plugged into the LMAs when the tasks are to be migrated and executed at the remote site. By doing so, the semantics of agent operations can be kept separated from the actual task the agent wants to execute. Furthermore, again different from the existing mobile agents, an LMA is able to carry multiple containers (i.e. multiple tasks to be executed) along its itinerary. By doing so, some network bandwidth could be saved if these tasks are to be executed on the same agent itinerary (when compared to the traditional approach of "one-task, one agent").

Although LMA itself does not contain the description of tasks it wants to execute, there are some fixed *basic agent operations* defined in the LMAs to support the normal agent activities. There are altogether four basic agent operations identified in the current design of LMA:

- **Migration.** The migration operation of the LMA determines the next site to be migrated according to its pre-planned itinerary (a kind of member state stored in LMA, which is in fact a list of trusted hosts), the current site LMA resides and the status of the participating environment. This operation is also allowed to adjust the pre-planned itinerary according to the environment change and the mission goal of the agent.

- **Communication.** The communication operation aims to deal with the interactions between LMAs hosted on the same site upon requests from the containers (tasks) being plugged in, at the same time hiding the details of the communication (e.g. the communication protocol used between LMAs) from them.

- **Security.** The security operation determines what kind of security measurements are needed to be enforced in accordance to its own security policy (another kind of member state stored in LMA) as well as the security policy enforced by the hosting site. Again, this operation is allowed to adjust the security policy according to the environment change.

- **Execution resumption.** As mentioned above, an LMA can carry several containers (tasks) with it along its itinerary. Therefore, execution resumption operations come in and determine which task are to be resumed on the destination hosts and how (the order) they are resumed.

From the programmers' point of view, LMA is in fact nothing more than a component carrying its associate data states and the descriptions of the code components (facets) it needs to use for executing the basic agent operations. By using the modified MA and COD (COD-MA) paradigms described in the previous chapter, the agent does not need to carry any codes when it migrates. When it needs to execute any facets that it requires to carry out basic agent operations, it will bring them in dynamically from the nearby sites and use them. The size of the agent can therefore be significantly reduced during migration, and therefore effectively reduces the bandwidth requirement when traveling from site to site.

Besides bandwidth saving, the COD-MA approach has also increased the flexibility and adaptiveness of the LMA. Since all the implementations of these basic agent operations are dynamically plugged in at run-time, they can be changed to adapt the environment as long as it stays in line with the mission of the agent. For example, a different migration operation can be plugged into the agent if the LMCS detects that there is a traffic congestion in the network along the agent's pre-planned itinerary. The pre-planned itinerary or even the migration strategy can therefore be dynamically changed upon detecting this change in environment. Similar claim applies to the security[2] and code resumption operations, where computation resource in this case is instead the limiting factor. For the communication operation, the flexibility comes from the fact that there is now freedom for two communicating agents to choose their own communication protocol, while this process is totally transparent to the tasks plugging into it (i.e. in this case, the agent itself has become an *adapter* of different protocols from the viewpoint of the container).

As a final word, it should be noted that for LMA, *no execution states* associated with it are needed to be maintained nor migrated. It is because the agent operations supported in the LMAs are expected to be *atomically executed* and *lightweight*. They have to be executed atomically since these basic operations are meaningful only when they are fully executed. For example, it does not make sense for an agent to migrate when it is on the way executing the routine that determines the migration destination, or when it is in the midway of the session communicating with other agents on the same site (e.g. it is waiting for other agents' response). On the other hand, the operations have to be lightweight in order to be executed quickly even under emergent migration event. Therefore, it is time- and resource-consuming (and also meaningless) to capture the execution states of the LMAs.

[2]Special case may be applied to the security operations, since it may involves two sub-operations that work hand-in-hand: encryption and decryption. The agent may therefore need some fore-planning (e.g. detecting the resource consumption status at the destination host) in choosing the facet suitable for it to perform the required operation.

In view of this, a weaker migration model should be enough to migrate the status of the LMA, in which only the data states of the agents (e.g. the agent's itinerary, the security policy it deploys, etc.) are migrated and the executions of the agents are always restarted from a pre-defined point at the destination site. In other words, the LMA can be readily migrated even on systems supporting only *weak mobility* (e.g. Java) without using any modification nor instrumentation schemes.

7.1.3 Container

Container is yet another active participant in supporting mobility in our architecture. As mentioned above, it provides a *task* abstraction to the LMA and is used to plug into the LMA to enable mobile code computations. Here, one may question about the need of container in our architecture, and wonder if *facets*, being code components, can already provide the task abstraction: facets, according to their definitions, should be readily used to plug into the LMA and perform the required functionalities. In fact, using facets alone is not enough to perform any *meaningful* computations or tasks. As mentioned before in Chapter 3, facets are stateless in nature. However, to execute a task, there are always some data and execution states generated during the computation process in order to describe the execution past for the future computation to base on. In order for the computation to be meaningful, these states must be stored somewhere. However, because of the stateless nature of the facets, these states information cannot be stored inside it and thus have to be stored elsewhere. This is where the container comes into the picture: it acts as a storage area to store the states that are useful for computation.

Besides acting as a *shared state holder* to make facet-based computations and tasks meaningful, the container also acts as a *facet holder*, which aggregates a set of facets implementing related functionalities that would work together to complete the specified task. This time, we again adopt the COD approach to store these "facets" to keep the container lightweight: instead of storing the actual facets, only the *facet specifications* describing the required functionalities are needed in the container. Also, it should be noticed that by the property of facets, we do not need to store all the facet specifications that are needed to perform the designated functionality inside the container. We only need to store the facet specification of the facet that is at the root of the facet execution tree for the designated functionality. Other facets required for this functionality would be loaded in during runtime on demand with the aid of the *Intelligent proxy system* in our architecture[3].

With both code and state components being able to be stored inside the container, the container

[3] Similar code-on-demand loading scenario applies to the basic agent operations for the LMA.

can now act as a task that can be statically run on a site[4]. However, in order to become a task that can be carried by LMA and support mobile code execution, the techniques on enabling migration support discussed in the previous chapters should also be built into the container. This brings out another responsibility that the container should take: the *support for state migration*. Besides storing the data state as discussed above, the container should also provide a place for storing instruction pointer state and stack state. Also, it should also play as the role of `sharedObj` for saving relevant thread states as that described in Chapter 5. In other words, the container is responsible to provide the `isMigrating` flag signifying whether migration is taking place to support *reactive migration*, and is responsible to store state tables like *lock table*, *wait set table* and *notify token table*, etc. and their associate functions in order to support thread state migration. Of course, this implies that all facets should have a variable `parentContainer`, so that before executing a facet, the facet loader of the client system would be able to set the reference of it to its corresponding container, in order to achieve the effect that the container is *shared* among threads in the multi-threaded scenario.

To the end, acting as a task may be useful only for interacting with LMA and computation, and most probably only be meaningful to programmers who have knowledge on how to program using the abstractions. However, it is meaningless to end-users, who have no knowledges on programming with container and facet abstractions, and do not know how to interact with facets to perform their desired functionality, since facets only have a programming interface to interact with different facets, but not end-users. In view of this, the container, during execution, can also provide a *user interface* (UI) for the end-users to access the facets inside the container. This can be done by linking the *UI events* triggered by the end-users to the corresponding *facet specification* inside container. Of course, which of the facet specifications that would be linked by the UI events depends on the UI programmers' design and the display capabilities of the target devices. Through providing the UI component, the container now not only acts as a task for the LMA to plug in, but also gives an application-like feeling to the end-users, thereby bridging the relationship between the end-users and the facets. In both cases, the container can be carried by the LMA to perform mobile computations.

7.1.4 The architecture

Figure 7-1 shows the an overall picture of our mobile code system architecture incorporating the three core entities described above.

[4]To make the container an active entity (i.e. executable), in our implementation, we declare the container to be a subclass of the `Runnable` class.

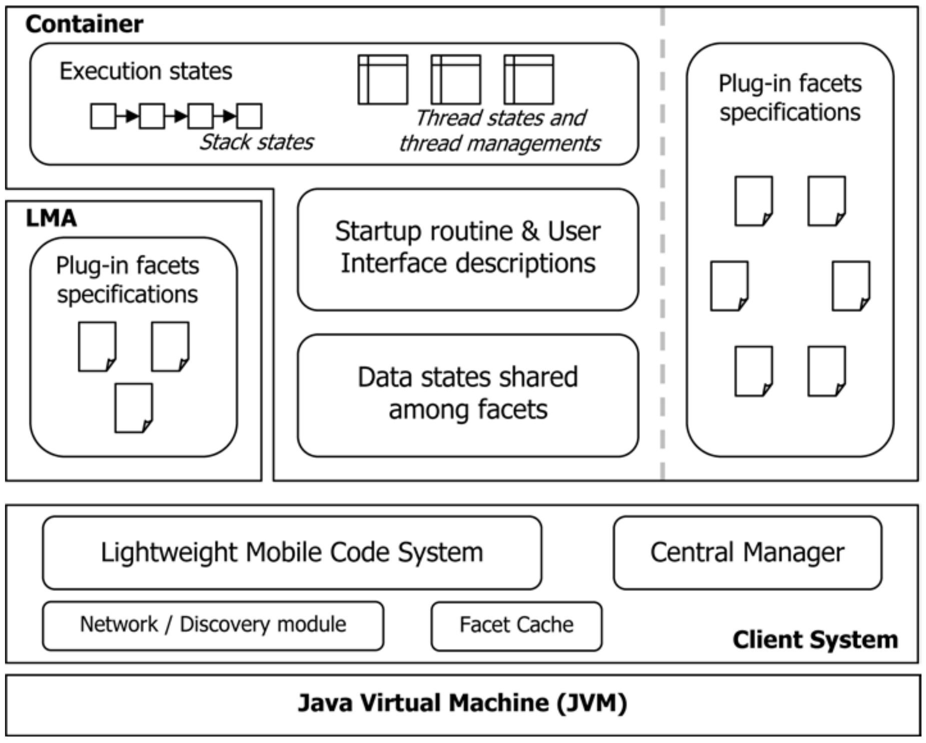
Container
Execution states
Stack states
Thread states and thread managements
Plug-in facets specifications
LMA
Plug-in facets specifications
Startup routine & User Interface descriptions
Data states shared among facets
Lightweight Mobile Code System
Central Manager
Network / Discovery module
Facet Cache
Client System
Java Virtual Machine (JVM)

Figure 7-1: Overview of our mobile code system architecture

It can be seen from the above architecture that both LMA and the container runs on top of LMCS of the client system, and all of them runs on top of the JVM for code execution. The container is mainly divided in three main parts. One is the *facet holder* area, which is the place where the facet specifications of the *root facets* are specified. Another is the *state holder* area. It stores the data states of the task/application being shared among the facets in the facet holder, as well as the execution states that are generated when the container runs. The final one is the *user interface*, where the routine controlling the UI and its relevant states are stored. On the other hand, as described before, LMA also contains the facet specifications for the basic agent operations. Basically, when migration takes place, the container would be plugged into the LMA, and the LMCS would then transfer the whole combined component to the migration destination.

7.2 The execution model of our system

In this section, we would briefly describe the execution flow between individual entities in the architecture during normal situation and when migration event occurs. We would also look into our design on the migration model.

7.2.1 The migration model

As discussed before, the basic migration unit in our system is LMA, with one or more containers plugged into it. In fact, in principle, the container can already be an individual migration unit. However, since our design essentially decouples the agent intelligence on migration, and container may need the LMA to communication with other agents, our design dictates that each container *must be* accompanied by one LMA upon migration. Despite this, the restriction may be loosen in the future if we find that the overhead of adding LMA to container in performing a task is too large.

On the other hand, it should be noted that our design is not able to support any migration units more fine-grained than a container (e.g. thread, a common migration unit for load balancing systems in cluster environment). It is mainly because of the absence of the object sharing support (e.g. a distributed shared memory system) in the global environment. Thus, even if it is possible for a thread to migrate to the remote site, it may not be able to access an object that is still residing on the local site.

Also, we have placed a restriction in our design that the container can be binded only to *one* LMA throughout its migration life time. It is just a design decision to make the procedure on tracking the task (carried by the container) and its relevant migration management easier and lightweight, rather than a scheme to improve the performance or enhance the overall features of the system. Since LMA is allowed to change its migration itinerary by its migration operation, the flexibility of the migration model would not be reduced even if such a restriction is placed. In order for this scheme to work, the LMCS is thus now responsible to keep the bindings between the LMA and their relevant tasks once they are established, until the LMA is retracted by its owner.

Another important attribute that should be taken into account in the migration model design is the *migration destination location*. To be specific, we need to determine which entities in our architecture are responsible to set the value of it. The resulting scheme is critical to the migration behaviour of the LMA: we need a location assignment scheme that would not affect the autonomy of the agent, at the same time allowing the container (task) programmer to control where the task should be migrated to.

We therefore adopt a *priority scheme* in setting the value of the migration location. Briefly speaking, in such a scheme, every entities in the architecture are authorized to raise the migration signal and set the corresponding migration destination location at any time. However, there may be a case that several entities want to set the location value simultaneously. In that case, the priority scheme would serve as a guidance on deciding which entity should be responsible to set the value of the migration location: the entity with the highest priority is granted to set the value.

Our current scheme on setting the priority of the entities is as follows: in general, the task executed in the container has the highest priority to set the location. It is because the container programmer is the one who knows the execution semantics of the task and is therefore responsible for setting the migration plan in accordance with the nature of the task. If the container programmer has no preference on the migration destination, the responsibility on setting the migration location goes to the LMA. With the goal and migration plan incorporated with the LMA, it should be able to make a correct decision on the migration destination that sticks with its goal. Finally, if the above two entities both fail to provide migration destination details, the LMCS will be responsible in setting the location[5]. In that case, the LMCS would not make any intelligent decision and simply set the migration location to those of some nearby sites. There is, however, one special situation in which the migration location must be overriden by it: when the migration is forced by the retraction request issued by the source LMCS (the LMCS where the LMA is deployed). The migration destination should not be set to any other values thereafter.

Figure 7-2 summarizes what has been covered so far in our mobile code system during normal execution. It presents a typical interaction model between various entities in the system during normal task execution.

7.2.2 Interaction model during normal execution

It all starts when the end-user runs the container. When the container starts to run, it will undergo a registration process (1): since the container itself is being run by an underlying thread, to prepare for thread state-capturing upon migration, it should be registered to get a `stateID` for the underlying thread. At the same time, a corresponding entry in the execution state tables of the container would be created for the thread. The real execution of the container can then start. The UI routine in it would first be activated to bring up the corresponding user interface (2), and the user can then interact with and execute a facet through that interface (3). When the user issues a facet execution request by issuing an event through the interface, the central manager of the client system would fetch and load copies of facets from either its local cache or an nearby proxy according to the specification of the container (4, 5). During the process of facet loading, the value of the member

[5]This situation usually happens when LMCS is the migration initiator, e.g. when emergent agent migration must take place because the security of the system is compromised or the system is running out of resource.

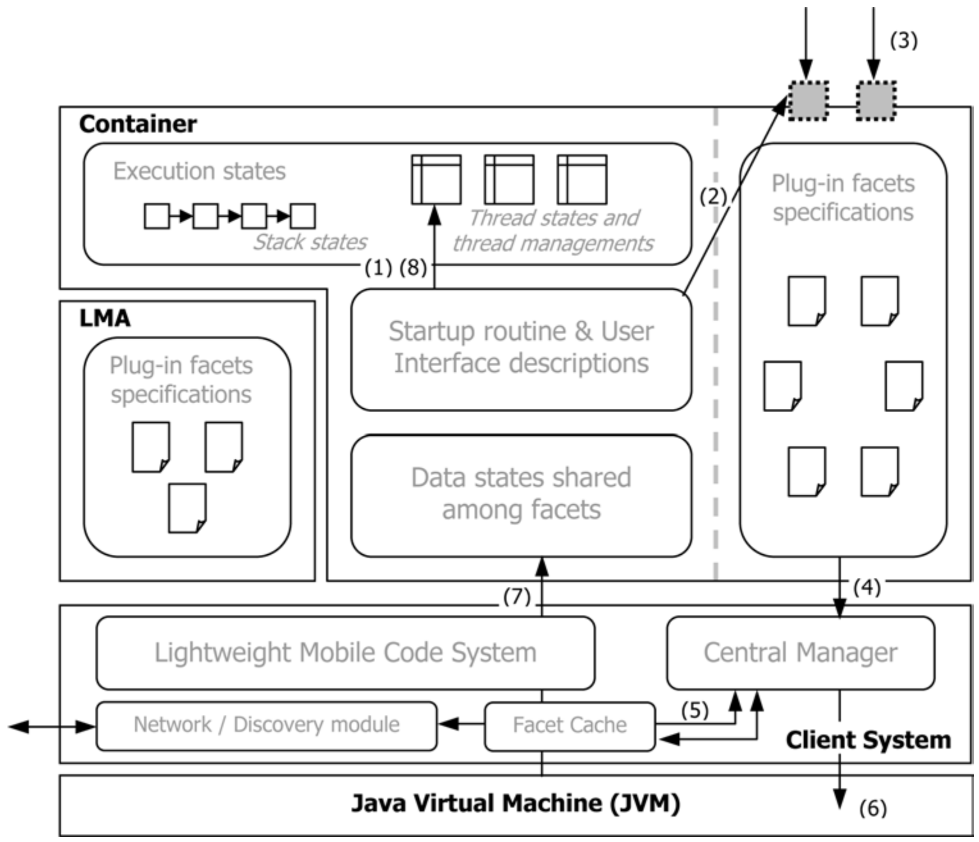
(3)
Container
Execution states
Stack states
Thread states and thread managements
(2)
Plug-in facets specifications
(1) (8)
LMA
Plug-in facets specifications
Startup routine & User Interface descriptions
Data states shared among facets
(7)
(4)
Lightweight Mobile Code System
Central Manager
Network / Discovery module
Facet Cache
(5)
Client System
Java Virtual Machine (JVM)
(6)

Figure 7-2: The interaction model of our mobile code system during normal execution

variable `parentContainer` of the facet being loaded would be set properly to point to the current container, and this value would be inherited to all the facets it invokes. As discussed above, by doing so, there is now a way for the current executing thread to capture and store its thread states into the shared container. After that, these facets would be executed (6), during which some shared data states are also saved into the container (7). When the execution is finished, a similar but reverse deregistering process would be held in the container, in which the corresponding entries in the state tables of the container are removed (8).

7.2.3 Interaction model during agent migration

Figure 7-3 continues to describe the interaction model between different entities in our system when migration takes place.

Suppose that we are on the way executing the facets in the container, and at the mean time, migration takes place. The event can be triggered either proactively or reactively. For the former case, the event is always triggered by the facet execution. For the latter case, it can either also be

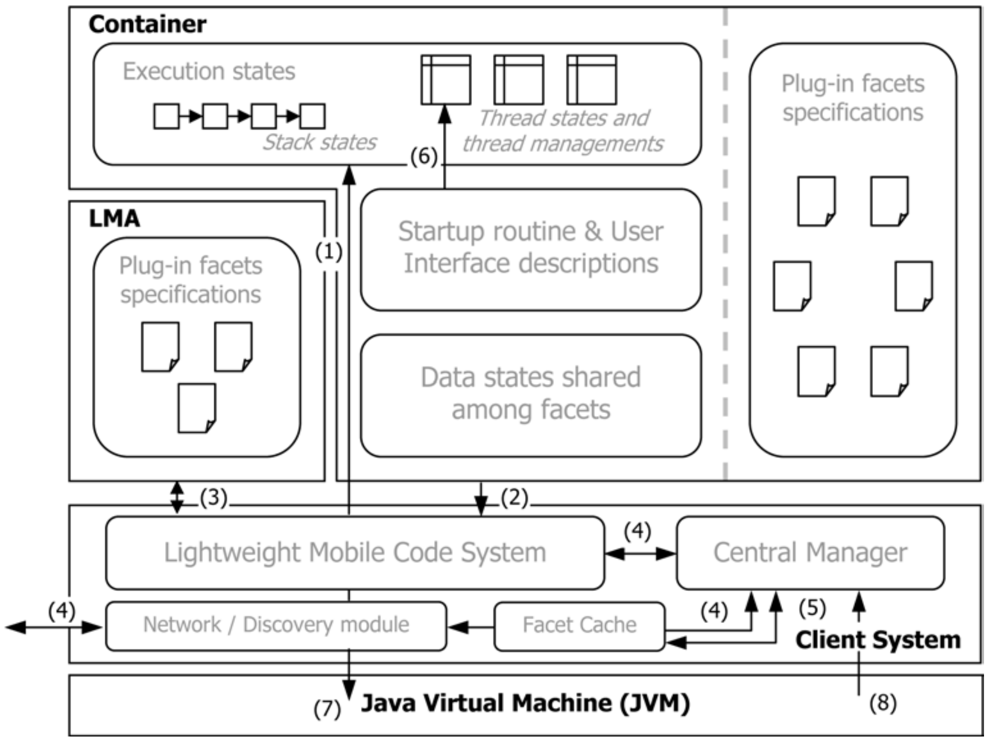
Container
Execution states
Stack states
Thread states and thread managements
(6)
Plug-in facets specifications
LMA
Plug-in facets specifications
(1)
Startup routine & User Interface descriptions
Data states shared among facets
(3)
(2)
Lightweight Mobile Code System
(4)
Central Manager
(4)
Network / Discovery module
Facet Cache
(4)
(5)
Client System
(7)
Java Virtual Machine (JVM)
(8)

Figure 7-3: The interaction model of our mobile code system during migration

triggered by the facet execution, or by the LMCS upon receiving an overloading signal from the central manager of the client system. In either cases, the container would be set into a *migrating* state. When this state is set, all the other threads that participates in the execution would trigger the state-capturing routine when they reach the migration checkpoints. The state thereby captured is then stored in the corresponding entry of the state tables inside the container (which was created when the thread register) (1). During the execution state packing process, the facet IDs of the executing facets are also stored together with the stack state. They are used to fetch the same facet again from the proxy at the destination site for proper execution restoration[6], or otherwise the proxy would return different facets, which makes the execution restoration incorrect.

When all the required states are successfully captured (i.e. when the state tables are full), the migration process starts. The LMCS would first check from the LMA-container binding table to see if there are any LMAs being assigned to the migrating container before (2). If not, the LMCS will create a new LMA for it. Otherwise, the old LMA will be used. After that, the migration destination location of the LMA would be set according to the priority scheme described in the previous section (3), and at this time, if necessary, the facets needed to be plugged into the LMA

[6]The state restoration process needs the same codes as those during the state capturing process.

would be fetched too (4, 5). On the other hand, if state-on-demand is supported, the stack state inside the migrating container will be chopped into several slices. The resulting execution states, together with the shared data states in the container, will then send to the destination site with the LMA according to the state-on-demand scheme described in the Chapter 6 (4, 5).

When the destination site received the LMA packed with the resumption states, the state resumption process begins. First, all the received objects are unpacked following the reverse order of the above process (i.e. 4, 3). It is noted that in resuming the execution of a container, the container should be authorized to be resumed according to the code resumption policy in the LMA. If it is not authorized, running that container is prohibited. Once a container is set to be in the *restoring* state, for each entry of execution states inside the state tables in the container, a corresponding new thread (computation component) would be created to take care of it. This new thread will then be re-registered in the container using the `stateID` that is carried together with the stack state (6). At the same time, it will send the list of facet IDs of the facets that are needed for resuming the execution to the proxy server for it to prefetch the required facets (4, 5). The facet execution of each individual threads can then be resumed using the state unpacking strategies described in Chapter 5 (7). If state-on-demand scheme is used in addition, the corresponding stack state segments will be brought in one by one using the scheme described in Section 6.4.2 (8).

7.3 Implementation of the design

The system and the entity interaction designs described in this chapter was already implemented using Java programming language. It works in co-ordination with the strong-mobility-enabled facets that are specially programmed and instrumented by our modified JavaGo source code preprocessor written in ML programming language. However, as stated in Section 5.10, our source code instrumentation scheme still exhibits some limitations on fully supporting strong mobility. For example, it cannot migrate resource states and user interface states. As discussed before, these states are either *fixed in nature* or not necessarily allowed to be migrated. In other words, they must be handled by other means. To deal with these cases, we follow the scheme that is commonly adopted by weak MCSs like Aglets and Voyagers: we rely on programmers to maintain these states they require to perform the desired functionality. We provide two `protected` methods: `onMigrate` and `onArrival` for the programmers to override, in which the contents of the `onMigrate` method would be executed just before migration, whereas the contents of the `onArrival` method would be executed immediately after arriving the destination site. By introducing these two methods, the programmers can, for example, use their self-defined protocols to continue reading a file through using the *network reference* strategy described in Chapter 4.

In addition, the current implementation of our mobile code system is just a proof-of-concept implementation of our mobility model. It is still far from being a full-fledged MCS with rich set of features support. For example, important features like *agent tracking* and *agent retraction* are absent in our current system design and implementation. It is mainly because in order to increase the availability of the scheme, the system designs and interactions involved in such strategies are usually too complicated to be implemented in our system, which is just used as a proof-of-concept on our mobility model (for example, the *shadow protocol* and its variants presented in [2] involve complicated interactions using time-outs between agents and sites). However, we believe that the schemes on supporting these features are readily plugged into our LMCS, since these technologies are orthogonal to our design on supporting the mobility model.

Besides including the agent tracking and retraction facilities, there are also some additional features that can be added into our implementation to increase the flexibility and power of the resulting agent programming model. For example, the current migration operation of the LMA only determines *where* the agent should migrate to. However, it does not decide *when* the migration should take place. Therefore, we may enrich the agent's ability on this aspect by including a *migration timeout* operation in the LMA for deciding when the agent should migrate. Also, the current design model does not allow a task (container) to dynamically deploy another task to a remote site by LMA for accomplishing its goal. This may weaken the power of the resulting programming model. Therefore, it is also regarded as one of the features that we have to incorporate into our design in the future.

7.4 Summary

In this chapter, we have introduced the architectural design of our mobile code system. We carefully described and analyzed the properties of three main entities: container, LMA and LMCS, in our architecture. We have also discussed how the strategies described in the previous two chapters in enabling strong mobility and making the system lightweight are incorporated into our design of the entities. After that, we presented an interaction model between the entities and the execution flows among them in order to enable flexible computation. Finally, we have looked into some of the inadequacies of our current implementation: although the current implementation is still far from being a real MCS that can be fully deployed and used, we believed that most of the features can be easily plugged into our system because they are mostly orthogonal to our design. In the next section, the above implementations of the strategies described in the previous two chapter would be analyzed and evaluated. Some experimental results would also be discussed.

Chapter 8

Evaluation and Discussion

In this chapter, we shall present the experimental results with our implemented mobile code system. These experiments show that our code instrumentation scheme can easily make the non-mobile multi-threaded codes be able to strongly migrate to another host after the instrumentation. Although the performance and the memory resource usage efficiency on executing the resulting code generally decrease after the instrumentation (as expected), they are still acceptable for most daily applications. On the other hand, the result of a simple test on the *state-on-demand* scheme shows a potential opportunity on reducing the bandwidth and memory usage on the target device.

We have conducted a few experiments on our implemented system using three common applications: an approximation of the value of π by evaluating the integral using rectangular rules, a simulation of the N-Body systems using the hierarchical Barnes-Hut method, and a simple Fibonacci program. The first two of them are multi-threaded programs that are used to evaluate the results of code instrumentation on multi-threaded programs, while the last one is just a sequential program which aims to give a brief insight on how the state-on-demand scheme can help to reduce the memory and bandwidth consumption. We will describe the experiments and discuss their corresponding results in the following sections.

8.1 The benchmarking environment

The experiments presented in this chapter are all conducted on a standard PC equipped with a Intel Pentium III 650MHz processor and 128M internal memory. The machine is running Linux operating system with 2.4.18 kernel. All the benchmarking programs are compiled by the development tools provided in Sun Java SDK 1.3.1, and run on its Java Virtual Machine with the JIT compilation enabled.

8.2 On evaluating the code instrumentation scheme

8.2.1 Benchmark applications

Up to now, there are no typical benchmarking applications specially for evaluating multi-threaded code instrumentation schemes[1]. In view of this, we have choosen two standard parallel applications that are commonly used in the literature of parallel computing: the parallel evaluation of the approximate value of π, and the simulation of the N-Body systems using the hierarchical Barnes-Hut method. As we shall see later, these two programs exhibit different execution behaviour that can help evaluating our instrumentation scheme on different aspects. We will briefly describe these two applications below.

Approximation of the value of π

This is a simple and direct parallel application that is commonly used as illustrations of parallel programs in many introductory books for parallel computing. The basic principle of the approximation is originated from the formula:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

By the basic principle of integral calculus, we know from the above formula that the actual value of π is in fact equivalent to finding the area under the curve $f(x)$ from $x = 0$ to $x = 1$, where $f(x)$ is given by:

$$f(x) = \frac{4}{1+x^2}$$

There are several numeral methods that are commonly used to find the area under a curve, e.g. Trapezoidal rules, Simpson's rules, etc. Among these approaches, we chose to use the simplest approach (yet with large approximation error): the *rectangular* rule. This approach is straightforward: the area under $f(x)$ in interval [0, 1] is divided in n strips. Each areas of these strips are then approximated by a rectangular strips with $f(a)$ as its height, where a is the mid-value along the sub-interval for that strip. The areas of the strips can then be summed up to get the approximate

[1]Some highly recursive programs, e.g. finding Fibonacci numbers, sorting n numbers using quick sort algorithm, etc. are the typical benchmark applications for evaluating the instrumentation schemes for sequential code. However, these programs are not applicable in the multi-threaded scenarios. Up to now, no benchmarking programs is available for the multi-threaded ones.

value of π. As one can see, parallelism can be achieved by distributing the calculation of strips' areas among parallel threads. *No inter-thread communication is needed*. The pseudo code below illustrates the idea described above, with the sub-intervals each threads are responsible to compute interleaved.

Pseudo Code 1 A subroutine for a thread to compute partial approximated result of π

```
1: $partialSum \leftarrow 0$
2: $n \leftarrow$ the number of strips
3: $stripWidth \leftarrow 1/n$
4: $myThreadID \leftarrow$ the thread ID assigned to this thread
5: for $i = myThreadID$ to $n$ do
6:     $x \leftarrow stripWidth * (i - 0.5)$
7:     $partialSum \leftarrow partialSum + stripWidth/(1 + x * x)$
8:     $i \leftarrow i +$ number of worker threads
9: end for
```

A simulation of the N-Body systems

N-Body problem is yet another well-known problem in the literature of parallel computing. In simple words, it is to compute the forces acted upon n *particles*(or *bodies*) through their self-interaction (e.g. gravatation forces, magnetic forces, etc.), so as to help us study about the movements of these particles, and also predict their future behaviour. Solutions of this problem contribute to many different areas, including astrophysics, molecular dynamics and fluid dynamics, etc.

In implementing the simulation, we adopted the well-known *Barnes-Hut approach*. Although a simple strategy to the problem, which is to accumulate forces $F(i, j)$ of particle j on particle i for every particles and advance the particles according to the accumulated force, can compute the result accurately and can be parallelized easily, the strategy is clearly inefficient (with computation complexity $O(n^2)$). The Barnes-Hut algorithm, on the other hand, solves the problem cleverly by using a divide-and-conquer approach. It uses a *quadtree* for representing the particles inside the two-dimensional space (or *octtree* for the corresponding three-dimensional case). Each nodes in the tree represents a *cell* enclosing certain area inside the space. A quadtree is constructed by recursively subdividing the root node of the tree, which represents the whole 2D space containing all the particles, into four nodes representing four sub-cells with equal sizes, until each sub-cell has *at most one particle*. Each cell contains the total mass and the position of the center of mass of all the particles in the subtree under it. A typical quadtree structure is illustrated in Figure 8-1.

Figure 8-1: A typical quadtree where no square contains more than 1 particle

Figure 8-2: The tree pruning idea in approximating the force calculation

After a quadtree (or octtree) is constructed, the tree is traversed from its root once per particle to compute the net force acting on it. The tricky point here for improving performance is that at each step of traversal, if the cell represented by that node is *well separated* from the particle, we can consider that the forces acted on the particle resulting from the particles inside that cell come from a single point of source. In that case, we just use the center of mass approximation to compute the force on the particle due to the entire subtree under that cell. Otherwise, if the cell is close enough, each of its subcells has to be visited. A cell is considered to be *well separated* from a particle if its size D, divided by its distance d of its center of mass from the particle, is smaller than a theshold θ, which is to control the accuracy of the approximation. Figure 8-2 illustrates this idea.

We can solve the problem using the Barnes-Hut algorithm in parallel. In the parallel code, there are five major steps in each iteration of computation. They are briefly described by the pseudo code 2 (Each statement inside the outer for-loop corresponds to a phase, i.e. line 02 is phase 1, line 03 to line 05 is phase 2, line 06 is phase 3, line 07 toline 09 is phase 4, and line 10 to line 12 is phase 5).

Pseudo Code 2 A typical flow of Barnes-Hut algorithm in parallel

```
 1: for each iteration of computing new positions of each bodies do
 2:     Build a QuadTree (or OctTree for 3D case) for all the particles
 3:     for each cell in QuadTree do
 4:         Compute the center of mass and totalmass of all the particles it contains
 5:     end for
 6:     Partition the particles among the threads to achieve load balancing effect
 7:     for each particle do
 8:         Traverse the QuadTree to compute the force on it
 9:     end for
10:     for each particle do
11:         Advance it to its next position using the force calculated
12:     end for
13: end for
```

We have implemented the above strategy in solving the *three-dimensional* N-Body problem. In our implementation, the array of information of particles and cells are shared among all the worker threads. *Barrier synchronizations* are therefore needed after phase 1, phase 2, phase 4 and phase 5 to ensure that all computations on every nodes for that phase have finished, so as to preserve the correctness of the overall computation. On the other hand, in usual cases, the phases in building the QuadTree (phase 1 and phase 2) are performed sequentially, because running in parallel slows down the execution. However, for our purpose of testing, we have made it parallel so as to create more potential synchronization points. Even if so, the performance of executing the resulting code is not greatly affected. As a final point of note, *no synchronization* is needed during the force computation phase (phase 4).

8.2.2 Reasons for choosing these applications

The reasons for choosing the above two applications as our benchmarking programs are a bit different from their typical uses in the current literature of parallel computing. In normal cases, these two programs are used to measure the performance of their hosting systems. It is because while the former application (computing the value of π) has no synchronization points, the latter one (the N-Body simulations) possesses abundant synchronization points, which are typically located at the barrier synchronization in between phases, and during the process of building QuadTree. The performance of the system under different synchronization scenarios can therefore be tested and compared.

These applications are also used in our system for comparing performance under different synchronization scenarios. However, performance is not a critical part of measurement in our benchmarking process. It is because performance is not a primary issue in the typical usage of mobile agents, as mobile agents are usually only involved in asynchronous communications. Also, in most situations, the hosting execution environment of the mobile agents possess only one processor. This makes the comparison of performance of multi-threaded program meaningless. Instead of measuring performance speedup, performance measurement in our experiment are just used to evaluate the impact on performance of inserting more thread-state saving codes into the normal codes, and to measure the performance slow down due to the new synchronization points introduced by our code instrumentation scheme. The more important issue for adopting these two applications is to compare the code size blown up and the size of the thread states that are stored and migrated during strong migration. These two measurements give an indication on how well our code instrumentation can be. Too large the size of code and transferred state are not acceptable. Finally, the complexity of the N-Body application can also be used to check the correctness of the resulting scheme, to see if the instrumented scheme is freely from deadlock situations under its intensive locking behaviour.

Because of the above reasons, the problem size of our benchmarking programs differs from

Program	Original	Instrumented	Growth
pi(150,000,000)	3240	15372	+374%
nbody(8)	43855	84676	+93%

Table 8.1: Growth in byte code size (in bytes) of codes instrumented by our scheme

those that are typically used for comparing and measuring performance. For example, in the N-Body situation, we used 8 particles instead of 128K particles, so that the threads would not be busy executing at the force calculation phase. This increases the chance of keeping threads to stay at the synchronization points (e.g. synchronization blocks, waiting states, etc.) during state capturing, achieving our aim of conducting the experiments.

8.2.3 Experiments and results

We have instrumented the two programs of the applications presented above and tested for the correctness of them. The executions and states of both multi-threaded programs can be correctly captured, migrated and resumed portably between two LMCSs running on *unmodified JVMs*. For the case of running the code for N-Body simulation, despite its high complexity and a huge amount of synchronization points introduced, our instrumented code can still be correctly and stably run and migrate to and fro between two different hosts for several hours without encountering any deadlock situations. On the other hand, the migration can also be triggered either *reactively* at any time upon LMCS's request, or *proactively* as the multi-threaded program requests. From this, we have proven the correctness of our instrumentation scheme.

Besides correctness of the scheme, we have also tested our instrumentation scheme in various aspects. The problem size of the π-calculation application we used for conducting the experiment is 150,000,000 (i.e. $n = 150000000$, where n is the number of sub-intervals), and that for the N-Body simulation program is 8 (i.e. there are totally only 8 bodies in running the simulation). The descriptions and the results of the experiments are presented in the following subsections.

Measuring the instrumented code size

We have measured the growth of code size due to code instrumentation. This code size is measured in its byte code form rather than in source code form, because Java codes are usually transferred in its byte code format. The experiment results are shown in Table 8.1.

As we can see from the table, the π-calculation program exhibits a high blow up in code size. It

Program	**Normal classes**	**Normal stack frame classes**	**Stack frame classes for `synchronized` block**
nbody(8)	69332	9751	5592

Table 8.2: Code size distribution (in bytes) of the N-Body program instrumented by our scheme

is mainly because in the program, the proportion of methods that need to be instrumented is much higher than that of the methods that need not to be instrumented. In that case, the size of code inserted dominates the size of the resulting codes, thereby resulting in high blow up of code size. In normal cases, for the programs with more complicated structure (e.g. the N-Body program), less proportion of code are usually needed to be instrumented. This results in a milder code size blow up.

We further investigate the increment in code size resulting from our instrumentation scheme. In this case, we look into the code size distribution of the instrumented N-Body code, as it provides a more realistic model for daily applications. The distribution is shown in Table 8.2. In the table, normal classes refer to the classes that are also present in the uninstrumented version of code, but these classes are being instrumented. The "stack frame classes" in the table are the stack frame data structure that are needed to store the required execution state. Here, the *normal stack frame class* refer to the ones that are added by the traditional JavaGo compiler for saving instruction pointer states and stack states, while the *stack frame class for the synchronized block* are the stack frame data structure needed by our scheme for saving the stack states of the `synchronized` block.

The code size blow up resulting from our instrumentation scheme mainly comes from two different sources: one is to add additional stack frame data structures for the `synchronized` block, and the other is to insert codes in normal classes. While the size of the former code is readily available as shown in Table 8.2, the size of the latter code is difficult to obtain from the interepted bytecode format, since the portion of the instrumented code becomes inseparable from the normal codes in the bytecode format. We therefore make an estimation by assuming that the ratio of uninstrumented portion to instrumented portion in the source code remains the same when they are compiled into the byte code. Under this assumption, we found that our code instrumentation scheme has added 35564 bytes for thread-state saving among the 124384 bytes of codes. In other words, our instrumented code takes about 29% in the normal classes, which is about 19.8K. With all these information, we can conclude that the code size blow up resulting from our scheme on thread-state saving is about:

		Number of threads			
Program	**Object transferred**	**1**	**2**	**4**	**8**
pi(150,000,000)	Container	3014	3270	3782	4806
	Container + LMA	3345	3601	4113	5137
nbody(8)	Container	31727	33182	34735	37830
	Container + LMA	32041	33497	35050	38145

Table 8.3: The size (in bytes) of the tranferred states under different scenarios by running the codes instrumented using our scheme

$$\frac{19824 + 5592}{43855} = 57\%$$

Notice that this result is independent of code size blow up resulting from the JavaGo's instrumentation scheme on saving stack state and instruction pointer state. In other words, among the total 93% code size blow up, 57% are contributed from our instrumentation scheme, while the remaining 36% are from that in JavaGo. The result is relatively acceptable, since common code instrumentation schemes (with supporting only instruction pointer migration and stack migration) would suffer from 50% to 120% of code overheads [37][38].

Measuring the transferred state size

Next, we measure the average sizes of states that we need to transfer to the destination hosts during migration. These sizes are measured by dumping the objects to be transferred into a file. Its corresponding size is then taken from reading the file size.

In the experiment, we measured several sets of state sizes by running the applications using different number of worker threads. Two set of values were then taken for each of the above cases: one is for measuring the sole size of the container during migration, while the other is for measuring the size of the container together with an LMA. Table 8.3 presents the experiment result.

From the table, we can see that the size of the states transferred to the remote site increases as the number of threads increase. These result meets our expectation, since increasing the number of threads implies that we need to keep additional states in order to resume the execution of these newly added threads. These states introduced by the newly added threads are mainly comprised of two different components: one is to store the information instruction pointer states and stack states of the newly added thread, while the other is to store the information about its thread state, i.e.

whether it was participating in some inter-thread communications before migration.

Looking closer to the table, we can find that some of the experimental results presented in it are in fact accountable. As we know from the description of application nature in previous section, the π-calculation program does not have any synchronization points. Therefore, we expected that all the states that are captured would be just the stack states and instruction pointer states of the newly added threads. No thread states would be captured. Indeed, this hypothesis can be proved correct by examining the difference between each successive experiment result as the number of threads increases: *exactly* 256 bytes of addition states were captured for each thread being added to the execution, and this captured size is independent of the time of when the state capturing occurs. The situation is different for the N-Body simulation application. Since synchronization activities frequently occurs during execution, a considerable size of states are comprised of the thread states. As a result, the size of states being captured in this case, as expected, are non-uniformly increasing as the number of threads increases, and they are dependent on when the state capturing is being carried out (Different time of capturing indicates that there are different number of threads waiting on the locks, which increases the thread states).

Also, from the above results, one can see that the overall growth in size of states being transferred is acceptable. For the N-Body application where a large amount of data states (mainly member states, which stores the information about the particles) have to be maintained, the increment in states is just about 3% to 4% per thread. For small applications where only little or no data states are stored, the increment in states also falls in a tolerable range of about 9% to 10% per thread.

As a final remark from the table, we can see that the size of LMA is typically small (about 310 to 350 bytes, depending how many containers and the complexity of the itinerary it carries). But of course, it is just a measurement of the size of data state in LMA (LMA has no execution states). The size of its code is not included.

Measuring the performance of the instrumented code

Finally, we measured the performance of our instrumented code. This time, we measure the time needed to run the π-calculation program with $n = 150000000$ on one processor. To prevent the execution of the program from monopolizing the processor, we force the executing thread to pause temporarily and allow other threads to execute (by invoking a Java method `Thread.yield()`) after computing for every 250000 subintervals. Therefore, the performance result does not reflect the real performance of the program. However, since we just need to see the impact on performance

		Number of threads			
Program	**Nature**	**1**	**2**	**4**	**8**
pi(150,000,000)	Uninstrumented	12389	16620	18786	20851
	Instrumented	12548	12736	12474	13163
nbody(8)	Uninstrumented	1908	2000	2403	2723
	Instrumented	2450	2888	3942	4944

Table 8.4: The average time (in ms) for running the uninstrumented and the instrumented codes for two applications using different number of threads

after instrumenting the code, doing so would not affect our analysis. On the other hand, for the N-Body application, we measure the time needed to advance 8 particles for 400 times (i.e. iterate the main loop presented in pseudo code 2 for 400 times). The average times in running the applications in both their instrumented and uninstrumented form using different number of threads are presented in Table 8.4.

Surprisingly, we can see from the table that the instrumented code for the π-calculation program outperforms the uninstrumented one. However, it may not reflect the real situation: as we have just mentioned, running the multi-threaded program on a single processor and the use of `Thread.yield()` may produce unpredictable performance result, since the performance may vary due to different underlying thread scheduling policies. This is especially true for computation intensive programs like the π-calculation program, where the effective use of the CPU cycles is a critical issue in determining the performance. If the underlying thread-scheduling policy always switches thread execution, it would greatly degrades the resulting performance of the program. The results for the π-calculation code may exactly experience this problem. In contrast, for the N-Body simulation program, the instrumented code runs slower than the uninstrumented ones as expected. It is because many barrier synchronization points have been inserted in the code to force the threads to execute in the pre-defined ways. This makes the processor usage more predictable for both cases, and the outcoming result is therefore more reasonable (i.e. more codes inserted implies slower code execution).

Another important information that has been brought by the experimental results is the performance degradation resulting from inter-thread synchronization due to our code instrumentation scheme. As one can see from the table, for the case of executing the π-calculation program where no inter-thread synchronization are introduced, the time needed for running the instrumented code remains more or less constant as the number of threads increases. However, for the case of executing the N-Body simulation program, the time needed for running the instrumented code increases at a higher rate than the uninstrumented one as the number of threads increases. This result indicates that *some new performance bottleneck* has been introduced in our code instrumentation scheme, and this bottleneck is related to inter-thread synchronization. Indeed, as we have mentioned before in the future work section of Chapter 5, the *shared object* used in our code instrumentation scheme is the potential performance bottleneck. It is because every thread that needs to perform synchronization, no matter which lock they are asking for, have to serially access the shared object before they can perform the corresponding synchronization activity. This, as shown in the experiment result, has become the scalability bottleneck: performance rapidly degrades as the number of threads increases.

All in all, despite that fact that the performance measurement is not very accurate, the performance degradation of our scheme is estimated to be around 28% for single-threaded program, and increases at a rate of around 8% per thread added. Again, compared with the average performance of other code instrumentation schemes only for capturing stack states and instruction pointer states (around 20% performance overhead), the performance of our code instrumentation scheme is again reasonable [37][38].

8.3 On evaluating the on-demand schemes for lightweight computations

We did not conduct many tests on evaluating how lightweight an execution can be made by our mobile code system. It is mainly because of the high flexibility and dynamicity provided by our mobile code system. In our system, the amount of bandwidth and memory saved are originated from either the reduction in the movement of codes and states by the on-demand scheme, or the reduction in the size of code being brought in from the Intelligent proxy system, which will adaptively choose the code suitable for the target execution environment. In other words, for codes with different execution status being run under different execution environment, the amount of bandwidth being saved are also different. It is therefore difficult to design a fair experiment to show how many bandwidth our scheme can save – the results differ from time to time.

In fact, one of the reasons why typical Code-On-Demand schemes (as adopted in the dynamic classloading feature) can reduce bandwidth consumption is that well-designed code components (such as our facet abstraction) has *no* interdependencies in between them, except the contracts of their invoking interface. In other words, these codes components can be replaced or dynamically loaded without introducing any additional memory or bandwidth overheads (although at an expense of execution performance loss). However, the situation is different for the case in our designed *state-on-demand* scheme. As we mentioned in Chapter 6, because of the inter-dependencies of stack frames and the design of the Java serialization mechanism, some overheads (i.e. objects that are shared between shared frames) must be introduced if we chop the stack frames into several fragments and transfer them on demand. In view of this, in this section, instead of evaluating how our system can make the mobile code execution lightweight, we will turn to investigate the effects of these overheads on lightweight executions, and show how it is possible that a lightweight execution can be obtained through the state-on-demand scheme.

8.3.1 Application used to test the state-on-demand scheme

To fully test the behaviour of the state-on-demand scheme, we need to find an application that would create a high stack of execution frames, so that the effect of frame segmentation on making the execution lightweight can become more obvious. Also, it should not be too complicated so that the analysis on our scheme can be simple, direct and illustrative. In view of these, we have chosen a typical recursive program as our benchmarking application: a program to calculate *Fibonacci number*. As one may know, the Fibonacci number can be defined by the following recurrence relation:

$$Fib(n) = \begin{cases} 0 & n = 0 \text{ or } 1 \\ Fib(n-1) + Fib(n-2) & n \geq 2 \end{cases}$$

It should be noted that in writing the code for finding the Fibonacci number, we did not use any performance optimization techniques, e.g. dynamic programming. The execution of the application can therefore follow the typical in-order execution tree traversal, which resembles the execution pattern of most normal codes.

8.3.2 Experiments and results

Before the experiments are carried out, we instrumented the Fibonacci program we have written, so that the stack states of it can be easily manipulated. As we have not designed nor implemented the code instrumentation engine to automatically (transparently) enabling the support of state-on-demand features in the code, we have to modify the instrumented code ourselves following the

Request round	0	1	2	3	4	5	6	7	8
Without state-on-demand	2211	-	-	-	-	-	-	-	-
With state-on-demand	1581	268	268	268	268	268	268	268	1309

Table 8.5: The size of state transferred(in bytes) at each round of frame request

steps described in Chapter 6. After ensuring that the program works as expected, we carry out the experiments presented in the following subsections by running the code through invoking $Fib(35)$. It should be noted that in these experiments, we *only* record the corresponding data when the height of the stack is 17 during migration.

Measuring the sizes of migrating objects

In this experiment, we measure the total bandwidth usage in transferring the whole execution from one host to another. We conduct the experiment using two different state migration schemes, one for migrating the execution state without using state-on-demand scheme, while the other for migrating the state using the state-on-demand scheme, with the 17 stack frames being chopped into 9 segments, each of which having 2 stack frames (i.e. a division of 2-2-2-2-2-2-2-2-1 is used, with 1 as the bottommost stack frame). The experiment result is shown in Table 8.5.

There are several findings from the experiment results. First, it is noticed that the size of the first segment and the last segment during the transferral is larger than the rest of the transferred segments. It is because the former segment is in fact the container that contains all the required data states necessary to resume the execution at the remote site, and the latter contains a reference pointing to the container (which is unavoidable since this state segment represents a method inside the container class). As we have discussed in Chapter 6, since these two segments are transferred in two separate sessions, although they are referring to the same container at the source site, the container has to be transferred twice in this situation because of the Java serialization mechanism. In that case, the latter segment is considered to be the *overhead* in the state segmentation scheme.

Second, it is noticed that the intermediate segments are equal in size. It is because in such a simple application as finding the Fibonacci number, the stack frames are usually used to store some simple data types (e.g. int, float, etc.) instead of the highly structured objects. In that case, the size of stack frames, and thus the state segments, are always the same. However, it should be realized that in real world situation, the size of state segment will vary as it may contain references to different objects.

Thirdly, from the result presented above, it may seem that the state-on-demand scheme would not save the overall bandwidth: the overall size of the segments transferred using state-on-demand scheme shown in the above experiment is 4766 bytes, which is more than 2 times larger than that when transferred all the required the segments in one shot. However, the above experiment only consider the case when the mobile agent travels one hop from its source site. It does not take into account the case where the mobile agent can travel multiple hosts. Consider the following example: the mobile agent for the above application has migrated for 5 times before the execution needs to use the topmost stack frame. In that case, for the migration that does not use the state-on-demand scheme (which has to transfer all the required stack frames to the destination sites), it needs to carry at least $2211 \times 5 = 13266$ bytes throughout the whole journey. However, for the migration using the state-on-demand scheme (which only has to transfer the topmost two frames), it only needs to carry the corresponding $1581 \times 5 + 268 \times 7 + 1309 = 11090$ bytes, which is less than that for the former case. Also, it follows from the above result that the bandwidth saved is expected to grow as the mobile agent become more active (i.e. it migrates more frequently), or the resulting state segmentation scheme can make the chopped state segments move more *lazily* on demand.

As a final remark, since the resulting segmentation scheme needs only the topmost two stack frames in order to carry out the execution, it effectively saves the memory usage for the target device from storing the unused bottom part of the stack. Co-operated with the Code-On-Demand scheme which downloads code that creates smaller stack frame sizes, the lightweight execution on the target device can therefore be achieved.

8.3.3 Comparing the effect of different state segmentation schemes

It remains to show that different state segmentation schemes can potentially make the chopped state segments move more lazily in general: as we learn from the previous section, the longer the stack frame can stay in its original site, more bandwidth usage can potentially be saved. In other words, we need to find a state segmentation scheme so that the bandwidth usage can be minimized.

In this experiment, we evaluate four different state segmentation schemes on chopping 17 stack frames: two of them are uniform segmentation schemes, *2-cut* (which results in 2-2-2-2-2-2-2-2-1 setting) and *4-cut* (which results in 4-4-4-4-1 setting), while the other two are non-uniform segmentation schemes, *triangular cut* (which results in 5-4-3-2-1-1-1 setting) and *exponential cut* (which results in 8-4-2-1-1-1 setting)[2]. The time interval between each successive pair of requests

[2]Here, one can see that there is an advantage of using 17 stack frames as the candidate of testing. In all the segmenta-

		Request Interval							
		1	2	3	4	5	6	7	8
2-cut	Size	268	268	268	268	268	268	268	1309
	Interval	81	105	152	529	2490	5785	23996	61885
	Cumulative	81	186	338	867	3357	9142	33138	95023
4-cut	Size	332	332	332	1373	-	-	-	-
	Interval	155	542	6699	85435	-	-	-	-
	Cumulative	155	697	7396	92831	-	-	-	-
Tri. cut	Size	332	300	268	236	236	1405	-	-
	Interval	213	1706	6739	22872	25151	36803	-	-
	Cumulative	213	1919	8658	31530	56681	93484	-	-
Exp. cut	Size	332	268	236	236	1501	-	-	-
	Interval	979	7551	22843	23943	38207	-	-	-
	Cumulative	979	8530	31373	55316	93523	-	-	-

Table 8.6: The time-intervals (in ms) between each pair of successive segment request and its corresponding cumulative time in different schemes, and their associate sizes of the state segments (in bytes) requested

Figure 8-3: The comparison of different segmentation schemes

for new segments for each of the above schemes was recorded. Table 8.6 shows the experiment results.

With the data collected in the above table, we used a simple strategy to evaluate the *laziness* of the chopped frames resulting from each of the segmentation schemes. Since the *cumulative time* shows what time a frame is required and is loaded in, it can be used as an score indicating how lazy it is: the higher the score of a frame is, the lazier it is. It follows that if the state-segmentation can achieve the highest sum of scores over all the transferred stack frames, it will be the laziest scheme. Of course, for fairness, the score is expressed as a percentage of the total execution time. The comparison result is shown in the Figure 8-3.

From the comparison, we can see that non-uniform segmentation schemes are usually lazier than the uniform segmentation schemes. It is because the time in between successive segment request increases exponentially as the execution moves towards the bottom of the stack. Non-uniform segmentation scheme benefit from this by transferring less frames as the execution needs the frames at the bottom of stack. This effectively increases the laziness of the scheme.

It should be noticed that the above comparison is just a simple test. In modelling the real

tion schemes described above, the last segment would only contain 1 stack frame, which finishes its execution and exits the program as soon as it is fetched. As one will see later, this makes the evaluation process easier.

situation, many new considerations are needed to take into account. For example, in the above experiment, although the triangular cutting scheme is lazier than the exponential cutting scheme, since exponential cutting scheme issue less segment requests and thereby produce less overheads (at the same time achieve similar degree of laziness when compared with the triangular cutting scheme), the exponential cutting scheme may result in having better efficiency in terms of execution performance and bandwidth usage. Also, the model presented here using the Fibonacci program is over-simplified. In real situations, the size of the stack frame varies as there may be different objects associated with the stack frames. In that case, the bandwidth optimization scheme may also need to take this size-variation into account. Also, the migration policy of the mobile agent adds another dimension to the problem: different migration policy may influence the effect of different segmentation schemes. Essentially, all these issues create new opportunities and open up new research area on issues of bandwidth saving and making the execution lightweight.

8.4 Conclusion

From the experiments described in this chapter, we have proved that our code instrumentation scheme is successful in enabling strong mobility in multi-threaded codes portably over the standard Java platforms so as to achieve overall computation flexibility. Through the experiments, we have also shown that our scheme in instrumenting the code does not greatly degrade the efficiency of execution in terms of bandwidth and memory usage, the instrumented code size, as well as execution performance. All these degradable are tolerable to most mobile applications, and are comparable to various code instrumentation mechanism existing in the current literature.

On the other hand, through a simple experiment, we have also shown how it is possible to reduce the amount of resource usage in an execution and make it lightweight through the use of state-on-demand scheme (and its corresponding code-on-demand scheme). We have also verified that the bandwidth usage can potentially be reduced further by striking the correct balance between the state segmentation scheme and the mobility policy of the mobile agent. All these strategies in making the execution lightweight, together with the computation flexibility brought by enabling strong mobility on multi-threaded programs, adds two bonus features to the existing mobile agent / mobile code technologies. It makes the relevant technologies of mobile agents more favourable to survive in the pervasive computing world.

Chapter 9

Conclusion and future work

9.1 Conclusion

Pervasive computing Its presence makes many data generated.

It is a new and promising research field that is still looking for a a commonly accepted background of concepts, methodologies and techniques.

Find flexible way in manipulating data Mobile codes (and mobile agent) is a way to address this.

But traditional mobile agent is rather heavyweight in some sense. It carries all the code and state that are required for its execution. This approach therefore is not lightweight enough Also, the

Our research addresses this by

9.2 Future work

On the code instrumentation scheme: - Hopefully can relieve the bottleneck from accessing the shared object - Supporting other multithreading constructs better.

On implementing the state-on-demand scheme: - Currently only a brief prototype. - A code instrumentation scheme for this is needed to design and implemented

LMA control interface: - Now LMA cannot be controlled by users, and is merely a transportation tool. Add in intelligence. LMCS management tool

- Hopefully the agent interface can be similar to that of aglet. - (2/5/2002) similar to Grasshopper.. MASIF... seems to be better

On enriching the features of LMA and LMCS: - to keep it in line with the current technologies. control algorithms, security, intelligence

Bibliography

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, third edition, 2000.

[2] J. Baumann. *Mobile agents: control algorithms*. Lecture notes in computer science; 1648. Springer, 2000.

[3] 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.

[4] F. Hohl. A Model of Attacks of Malicious Hosts Against Mobile Agents. In *Fourth Workshop on Mobile Object Systems: Secure Internet Mobile Computations*, France, 1998.

[5] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., 1999.

[6] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[7] Dale Green. The Java^{TM} Tutorial: The Reflection API. `http://java.sun.com/docs/books/tutorial/reflect`.

[8] Robert Grimm, Janet Davis, Eric Lemar, and Brian Bershad. Migration for pervasive applications. Submitted for publication.

[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. Submitted for publication.

[10] Hasan, Jürgen Jähnert, Sebastian Zander, and Burkhard Stiller. Authentication, Authorization, Accounting and Charging for the Mobile Internet.

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

[12] IKV++ Technologies AG. Grasshopper 2: The agent platform. http://www.grasshopper.de.

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

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

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

[16] Matchy J. M. Ma, Cho-Li Wang, and Francis C. M. Lau. JESSICA: Java-enabled single-system-image computing architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.

[17] Ricky K. K. Ma, Cho-Li Wang, and Francis C. M. Lau. M-JavaMPI: A Java-MPI Binding with Process Migration Support. In *CCGrid*, Berlin, Germany, 2002.

[18] Microsoft. Microsoft .NET architecture. http://www.microsoft.com/net/.

[19] Mount Laurel. Bluestone software contributes freeware to XML/Java Movement. http://industry.java.sun.com/javanews/stories/story2/0,1072,11559,00.html.

[20] Nalini Moti Belaramani. A Component-based Software System with Functionality Adaptation for Mobile Computing. Master's thesis, The University of Hong Kong, 2002.

[21] George Necula. Proof-Carrying Code. http://raw.cs.berkeley.edu/pcc.html.

[22] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, Berlin, Germany, 1997.

[23] S. Loureiro. *Mobile Code Protection*. PhD thesis, ENST Paris / Institute Eurecom, 2001.

[24] Takahiro Sakamoto, Tatsurou Sekiguchi, and Akinori Yonezawa. Bytecode Transformation for Portable Thread Migration in Java. In *ASA/MA*, pages 16–28, 2000.

[25] T. Sander and C. Tschudin. Towards Mobile Cryptography. In *1998 IEEE Symposium on Security and Privacy*, Oakland, California, 1998.

[26] S.Bouchenak. Making Java Applications Mobile or Persistent. In *Proceedings of the COOTS'01*, San Antonio, Texas, USA, 2001.

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

[28] John Shandor. GRID today: Grid computing – Today and tomorrow: another view. `http://www.gridtoday.com/02/0812/100221.html`.

[29] 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.

[30] Guiherme Soares and Luis Moura Silva. Optimizing the Migration of Mobile Agents. In *Mobile Agents for Telecommunication Applications MATA'99*, Ottawa, Canada, October 1999.

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

[32] Stefan Fünfrocken. Transparent Migration of Java-Based Mobile Agents. In *Mobile Agents*, pages 26–37, 1998.

[33] Steven D. Gribble, Matt Welsh, Rob von Behren, Eric A. Brewer, David Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R.H. Katz, Z.M. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. To appear in a Special Issue of Computer Networks on Pervasive Computing.

[34] Sun Microsystems. Sun Open Net Environment (Sun ONE) architecture. `http://wwws.sun.com/software/sunone/`.

[35] N. Suri. An Overview of the NOMADS Mobile Agent System, 2000.

[36] 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.

[37] Torsten Illmann, Michael Weber, Frank Kargl, and Tilmann Krüger. Transparent Migration of Mobile Agent Using the Java Platform Debugger Architecture. In *Proceedings of the MA'01*, Atlanta, USA, December 2001.

[38] Eddy Truyen, Bert Robben, Bart Vanhaute, Tim Coninx, Wouter Joosen, and Pierre Verbaeten. Portable Support for Transparent Thread Migration in Java. In *ASA/MA*, pages 29–43, 2000.

[39] Vivien Wai-Man Kwan. A Distributed Proxy System for Functionality Adaptation in Pervasive Computing Environments. Master's thesis, The University of Hong Kong, 2002.

[40] W3C. Web Services. `http://www.w3.org/2002/ws`.