Design and Implementation of concepts for supporting component migration based on OSGI

Student project in Computer Science
Presented to Computational Engineering program of TU Dresden

Vu Duc Lam
November 1st, 2006-April 30th, 2007

Supervisors:
Dr. Thomas Springer
Dipl. Inf. Kay Kadner
Chair of Computer Network, TU Dresden
Abstract

Mobile code is an emerging and promising approach to distributed computing in recent years. With mobile code paradigm, software component can be obtained from remote systems, transferred across a network, and then downloaded and executed on a local system with or without explicit installation or execution by the recipient. Over just the past few years OSGi has been specified by the OSGi Alliance (formerly known as the Open Services Gateway initiative). OSGi technology is a component integration platform with a service-oriented architecture and life-cycle capabilities that enables dynamic delivery of services. OSGi is an universal middle-ware. These capabilities increase the value of a wide range of computers and devices that use the Java platform. This thesis discusses many principles and aspects of mobile code and combines them with the dynamic service platform OSGi. The combination might increase original capabilities of OSGi in many facets, namely to support migration concept of OSGi components. The lightweight system has been elaborated as an extension part of the original OSGi platform to allow migration, distribution and collaboration of OSGi components among different OSGi platforms. The system is built on top of Java Remote Method Invocation (RMI). It provides flexibility for a broad range of applications that can be implemented on top of it.
Acknowledgement

This report is a result of my student project (Grosser Beleg) provided by the Chair of Computer Network at TU Dresden.

I would like to take the opportunity to present my sincere thanks to Dr. Thomas Springer at the Chair of Computer Network, TU Dresden for giving me the unique opportunity and the continual 6-months guidance and encouragement. Special thanks go to Dipl.Inf. Kay Kadner at the Chair of Computer Network, TU Dresden for valuable instructions during my implementation. My warm gratefulness and gratitude goes to Dr. Thomas Springer and Dipl.Inf. Kay Kadner for being my supervisors. Thank you all for teaching me a way of thinking and realizing a hard concept. And my grateful acknowledgement goes to you both from not only assist me in writing but also examine this paper.

This work is dedicated to my family and friends who have given me strong support during the period of implementing the project work.

Dresden, March 19th, 2007
Vu Duc Lam
# Table of Contents

Chapter 1 Introduction..............................................................................................................6
  1.1 Motivation.........................................................................................................................6
  1.2 Thesis objectives and applied scenarios.................................................................6

Chapter 2 Migration..................................................................................................................8
  2.1 Fundamental concepts of migration..............................................................................8
  2.2 Strong Migration..............................................................................................................9
  2.3 Weak Migration...............................................................................................................9

Chapter 3 Comparison between weak migration and strong migration.................................11
  3.1 Point in time for initialization......................................................................................11
  3.2 Complexity and Feasibility...........................................................................................11
    3.2.1 Strong Migration.......................................................................................................11
      3.2.1.1 Strong migration based on modification of Java Virtual Machine...................11
      3.2.1.2 JIT Recompilation..........................................................................................13
      3.2.1.3 Source code preprocessor..............................................................................13
      3.2.1.4 Strong migration based on byte code transformation.................................15
      3.2.1.5 Java Platform Debugging Architecture modification...............................17
      3.2.1.6 Continuation Concept and Javaflow.............................................................18
      3.2.1.7 Translating Strong Mobility to Weak Mobility............................................19
    3.2.2 Weak Migration.......................................................................................................20
  3.3 Size of data transfer......................................................................................................20
  3.4 Latency of migration.....................................................................................................21
    3.4.1 Strong migration.......................................................................................................21
    3.4.2 Weak migration.......................................................................................................22
  3.5 Performance................................................................................................................22
  3.6 Conclusion....................................................................................................................22

Chapter 4 Related works........................................................................................................24

Chapter 5 OSGi-basics and implementation.............................................................................26
  5.1 OSGi Technology ........................................................................................................26
    5.1.1 Overview about OSGi Service Framework.........................................................26
    5.1.2 Configuration Admin Service .............................................................................27
    5.1.3 HTTP Service .......................................................................................................28
    5.1.4 Service Tracker....................................................................................................28
    5.1.5 Package Admin Service ......................................................................................29
  5.2 Definition of OSGi component......................................................................................29
  5.3 OSGi implementation...................................................................................................31
    5.3.1 Open source implementation..............................................................................31
    5.3.2 Commercial implementation...............................................................................32

Chapter 6 Stateful weak migration solution for OSGi component..........................................33
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 General requirements</td>
<td>33</td>
</tr>
<tr>
<td>6.2 General Design</td>
<td>33</td>
</tr>
<tr>
<td>6.3 Comparison RMI with Socket, Web Services and SOAP</td>
<td>35</td>
</tr>
<tr>
<td>6.4 System Architecture In Details</td>
<td>37</td>
</tr>
<tr>
<td>6.5 Interaction and co-operation between the system components</td>
<td>38</td>
</tr>
<tr>
<td>6.6 Solution for migration concept</td>
<td>39</td>
</tr>
<tr>
<td>6.6.1 Migration Decision</td>
<td>39</td>
</tr>
<tr>
<td>6.6.2 State Saving</td>
<td>40</td>
</tr>
<tr>
<td>6.6.3 Transferring of Code and State Information</td>
<td>42</td>
</tr>
<tr>
<td>6.6.4 Handling of OSGi bundles dependencies</td>
<td>43</td>
</tr>
<tr>
<td>6.6.5 Continuation of Executions and State Re-establishing</td>
<td>43</td>
</tr>
<tr>
<td>Chapter 7 Implementation</td>
<td>44</td>
</tr>
<tr>
<td>7.1 User Migrative OSGi Component</td>
<td>44</td>
</tr>
<tr>
<td>7.2 Migration Manager Component</td>
<td>45</td>
</tr>
<tr>
<td>7.2.1 Constituents of Migration Manager</td>
<td>45</td>
</tr>
<tr>
<td>7.2.2 Migration Manager Server</td>
<td>46</td>
</tr>
<tr>
<td>7.2.3 Migration Manager Client</td>
<td>50</td>
</tr>
<tr>
<td>7.2.4 HTTP Stub Downloader</td>
<td>52</td>
</tr>
<tr>
<td>7.3 Bundles Manager Component</td>
<td>53</td>
</tr>
<tr>
<td>7.3.1 Bundles Manager Functionalities</td>
<td>53</td>
</tr>
<tr>
<td>7.3.2 Class Diagram</td>
<td>54</td>
</tr>
<tr>
<td>7.4 Testing Results</td>
<td>54</td>
</tr>
<tr>
<td>Chapter 8 Conclusion and Outlook</td>
<td>58</td>
</tr>
<tr>
<td>8.1 Summary</td>
<td>58</td>
</tr>
<tr>
<td>8.2 Future Works</td>
<td>58</td>
</tr>
<tr>
<td>8.3 Personal Evaluation</td>
<td>58</td>
</tr>
<tr>
<td>Literatures</td>
<td>60</td>
</tr>
<tr>
<td>Appendix A: Code snippets</td>
<td>63</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

This chapter presents the overview of this thesis. Firstly, the motivation is introduced. Then the objectives and applied scenario of this thesis are outlined.

1.1 Motivation

The OSGi Service Platform is a runtime environment dedicated for dynamic component technology. The OSGi specification determines a standardized, component-oriented, computing environment for networked services. The OSGi Platform can be utilized in many distinct kinds of system from embedded software system for small networked devices in personal products to enterprises and it provides capability to manage the full life-cycle of software components in devices from every place in the whole network. Software components furthermore can be dynamically installed, updated or removed without being preventing the device from continuing its operation. A key element of this initiative is the OSGi framework, which is a lightweight framework for deploying and executing service-oriented applications [15]. Thanks to the management mechanism provided by OSGi, bundles can locate and invoke services offered by other bundles however unfortunately just in a single OSGi framework. Intuitively, OSGi lacks of sufficient supporting for application mobility among distinct OSGi frameworks. We all know that mobile code is an emerging and promising approach to distributed computing in recent years because of providing a capability to dynamically alternate binding between software components and locations where they are executed. Additionally, mobile code extends multiple capabilities of systems such as bandwidth-efficient communication, disconnected operations, and support for dynamic and flexible systems. Therefore, the paper comes up with an idea to construct a system that supports mobile code paradigm above of the underlying OSGi Framework. This approach perhaps can build a bridge to gap between distributed service-oriented architecture (SOA) applications and distributed service-oriented infrastructure (SOI).

1.2 Thesis objectives and applied scenarios

Under a scope of the student project, many steps have to be completed. First, many existing approaches to support Java object migration at many levels must be studied deeply. Subsequently, based on the researched results acquired from the first step, an investigation to verify if there is any existing approach which can be used in terms of supporting OSGi component migration. To implement the most feasible and appropriate approach is the next step. An evaluation of the overall achievements of the project under many aspects, namely performance, technologies, commercial usability and extendibility is the final step.

In this paper, a system architecture for fast migration of run-time OSGi components based on OSGi frameworks is proposed. The system, or the middleware in other words, should also satisfy some substantial requirements:

- Completely transparent with end users: End users do not have any wisdom about
implementation of migration concept and communication technologies.

- Communication technologies are partly transparent with application programmers. Additionally, the system is expected to optimize application programmers efforts as much as possible.
- Independence of heterogeneity of underlying systems: operating systems, hardware, Java Virtual Machines and different implementations of OSGi Platform.
- Good Performance.
- Support stateful migration of OSGi component: Stateful migration means that an OSGi bundle is migrated along with its saved states.
- Ability to be installed in resource-constrained devices: PC Pocket, Palm Top, PDA, etc.

The system is useful to be employed to build distributed applications on top of it that have the following targets: to support disconnected operation, to do computation in off-loading mode, communication among different hosts is performed in an unstable and not strongly managed network with limited communication bandwidth, to reduce network traffic. More in details, the application areas of the system can be illustrated out in the next following sentences. First, the middleware can be used to balance work load between different nodes in LAN or WLAN network. A network administrator can explicitly move some runtime OSGi components from heavy-loaded nodes to less-loaded ones. Second, the middleware is really useful in any system which needs to increase availability of OSGi component because an administrator can easily relocate a specific OSGi component in a machine that is going to be shut down or to be restarted. This routine makes sure that the OSGi component's business services are always reachable from other applications from inside or outside networks. While working in WLAN under limited bandwidth and high congestion condition, it is useful to transfer one OSGi component from one host to another place where it has local access to resources and it can locally process obtained data. Finally, in a single OSGi platform, services of an OSGi bundle can cooperate with services of other bundles through a sharing objects mechanisms provided by OSGi Service Registry. With the system, we can distribute OSGi components among multiple hosts but we can move OSGi components from one host to another host to cooperate with local bundles in order to set up complex business operations when it is necessary.
Chapter 2  Migration

This chapter gives a general view for the fundamental concepts of migration. Afterwards, two main classes of migration, namely strong migration and weak migration, will be specified.

2.1 Fundamental concepts of migration

Migration concept can be normally classified based on two fundamental aspects: state and code migration. Code migration refers to code transfer and state migration refers to an object's state transfer [1].

More in detail, code migration means that code of a Java object has to be transmitted from a source to a destination location. If the Java object especially does not consist of only one class but it handle references to many other objects, these objects codes have to be transmitted too. However, code migration should just transfer code of referenced objects when there is a need [1].

In other side, state migration comprises of many more aspects, namely execution and data migration. The execution migration is composed of further parts as program counter and thread migration. Program counter migration means that execution flows continues at the destination host at the same point when it is hang previously. Thread migration means that in order to migrate a Java object composed of many child threads, it must be put into consideration to capture and restore correctly these threads states. Data migration in turn is formed by stack, program member and resource migration. The migration of all member variables of a Java object is vital because member variables are places where a Java object store its computation results or query results, etc. Stack migration is used to transfer all local data containing of local variables and operands of computation in every method on a method's call stack. Moreover, stack migration theoretically relies on program counter migration. Finally, resource migration regards to external resource that a Java object may hold references to, particularly database connectivities, opened sockets or accessed files, etc. In theory, except a problem of migration local file access, all other problems can be brought down to migration of single or multicast socket streams [1].
2.2 Strong Migration

Figure 1: Classification of strong migration concept [1]

From figure 1, strong migration can be identified as a migration technique which can be realized by combining code migration as well as strong state migration. Strong state migration is extensively determined from a concept of state migration. The strong state migration is claimed to require strong execution migration and strong data migration. Strong execution migration in turn is an extended concept of the execution migration and it can be attained by implementing both strong program counter migration, strong thread migration. The strong data migration in turn demands stack, member and resource migration. Intuitively, strong migration is better for programs performing intensive computations in a long period, but it is much harder to implement than weak migration. In some scientific articles, people use the term “transparency migration” as a synonym of “strong migration” [1].

2.3 Weak Migration

Weak migration is the most straightforward concept of migration to implement. In weak migration form, only code segments of a Java object is transmitted to a target host with some initialized data. Or in other words, weak migration can be realized by only code migration at few points in time together with some initialized data.
Figure 2: Classification of weak mobility [31]

From the figure 2, mechanisms supporting weak mobility can be classified and grouped systematically according to the direction of code transfer, the nature of code being moved and the synchronization mechanisms engaged. As for the direction of code transfer, a mobile program can either be fetched by another executable program from the different side or to be shipped to another host. The code can be immigrated either as a stand-alone code or as a fragment code. Standalone code is self-contained to be used to instantiate a new execution unit. In contrary manner, code fragment can be joined in a context of some already running code and will be executed at the end. Weak mobility mechanisms can be also either synchronous or asynchronous [31]. Migration term also can be classified into two different approaches: active and passive objects. Basically, there is a difference between migration of active and passive objects. Migration of active objects deals with independently running software modules or executables like mobile agents.

In fact, there are maybe other types of migration that miss one or some of the mentioned and required aspects that must be provided with strong migration. Actually, we can simplify the problem of classifying different types of migration by assigning those kinds of migration into the weak migration category [1].
Chapter 3  Comparison between weak migration and strong migration

In this chapter, comparisons between strong migration and weak migration in many facets which can be named like that: point in time for initialization, complexity and feasibility, size of data transfer and latency of migration, will be made.

3.1 Point in time for initialization

Generally, better aspects of strong migration over weak migration are that a mobile program that contains long-running or long-lived threads may be unexpectedly transferred from one host to a destination host at any point of time. And then it can be restarted to execute exactly on the point where it is suspended at the original host. Basically, such a strongly transparent migration mechanism can enable a continued thread's execution without loosing any data in its ongoing execution [18].

In contradicts to the strong migration, weak migration may be constituted from two alternative forms of the weak migration, namely an initialization migration and a method migration. With the initialization migration, execution of a mobile object always starts at the same initial point, like Java Applet uses Init() method for any run. Programmers must store and restore a mobile object's states by their own efforts. These tasks can be done before and after a migration process really happens. However, obviously the migration process is not transparent any more to them. With method migration, programmers must determine in which methods the execution of mobile object should continue after the migration process is accomplished. In addition, programmers somehow have to pass some initialized data to these methods. Anyway, the weak migration can only be started to perform after completing all methods call currently invoked in a main process of the mobile program [18].

3.2 Complexity and Feasibility

The section is dedicated to illustrate out substantial differences between strong migration and weak migration concepts regarding to complexity and feasibility of their realizations.

3.2.1 Strong Migration

In the next few parts, each existing approach dealing with strong migration will be investigated deeply. Results of this study will give us a closer look at complexity and feasibility of each solution if we want to employ them to develop any system that supports strong mobility concept.

3.2.1.1 Strong migration based on modification of Java Virtual Machine

Approaches that modify JVM (such as Sumatra [8], JavaThread [9] [24]) often introduce a
problem of managing a Java Virtual Machine itself. In JavaThread [9], authors proposed an approach to extend standard Java Virtual Machine in order to allow capturing execution states of a Java mobile application by using the type inference and the dynamic de-optimization techniques [5].

The main objectives of this approach focus on completeness and performance of migration mechanism. Firstly, regarding to the completeness objective, in order to be able to capture entire states of a thread, they decided to implement the migration mechanism within a standard JVM. This is the only way to be granted full access to stacks of a Java thread. Essential points of this mechanism can be provided by this approach are that, with Java object serialization, thread serialization allows thread's execution states to be stored in some data structure and to be copied on a disk to implement persistence or to be transmitted to a remote machine for mobility purpose [5].

Regarding to extension of JVM, two prototypes are considered based on the Interpreter-based serialization (ITS) or on a basis of the capture time-based thread serialization (CTS) [5].

The first prototype is dependent upon a modification of Java interpreter, so that it can capture data types whenever any byte-code instruction saves its data in a method stack. However, this solution leads to a possible problem because it is not compliant with a standard JIT compilation. In many cases, it is not a realistic solution due to this non-compliance. In addition, it also introduces an execution performance overhead [5].

On the other words, CTS keeps execution performance away from overhead, but a cost of this solution is passed on the serialization latency. And thus, it is not recommended to be employed for any mobile system with the high frequency serialization requirements [5].

Finally, the adoption of a modified JVM can introduce problems of trust and security bugs. But its main disadvantage is that it tightly relies on an extension of a typical JVM. Hence, this thread serialization mechanism can therefore not be used on randomly existing virtual machines. Standard existing virtual machines can not be used. In other words, the major drawback of this approach is a lack of portability and a shortage of efficiency in comparison with other approaches at application level [5].

In fact, it is observable that a mobile network has a heterogeneous character. Therefore, it can not be ensured that all devices are running same kinds of JVM. Supposing that albeit we have a power to interfere and modify some things at JVM level but if we want to migrate code and execution states of a Java mobile program between two Java Virtual Machines, these two Java Virtual Machines must be identical. That means they have the same structures of stack, memory representations. For example, neither the Sun JVM version 1.4 and Sun JVM version 1.5 nor Sun JVM version 1.4 and JVM implementation of IBM are identical. And obviously this approach is an infeasible one in most cases because mobile network is heterogeneous. Therefore, it is impossible to make sure that Java Virtual Machine running on arbitrary devices are exactly alike. Especially, in some mobile device, JVM is a substantial part of its firmware and there is no way to modify its JVM. Besides that, it needs a lot of efforts to be contributed and time to be consumed to understand a technical specification of a standard JVM. Without a scholar knowledge about internal structures of the JVM, it is likely impossible to extend its implementation in anyway. So that, modification implementation of JVM in order to support strong mobility might not be a right way to deal with the typical problem of supporting OSGi components migration in the heterogeneous mobile
network.

3.2.1.2 JIT Recompilation

Three Chinese researchers, Wenzhang Zhu, Cho-Li Wang, Weijian Fang and Francis C.M. Lau, in the paper [11] declared a new approach that uses JIT recompilation technique to provide transparent migration of Java threads. A JIT compiler is a dynamic compiler being a portion of JVM. It interprets Java methods into native codes when the methods are first invoked. The main idea of this approach is to use JIT compiler to recompile the Java methods in a stack context to assist extraction and restoration of Java threads context at a migration time. Such actions are only called when a migration request is performed. In the paper, authors mentioned two concepts: Raw Thread Context (RTC) and Byte-code Oriented Thread Context (BTC). RTC consists of a program counter, native thread stack, hardware machine registers. BTC is a more portable context can be used to resume a thread’s execution in any node type. BTC is completely independent from machine hardware architectures, operating systems and types of JVM. At a point in time when a migration request is carried out, a thread is suspended and RTC of the thread is transformed into an equivalent BTC. The BTC will then be moved to a target node where it will be converted back to the original RTC. And the thread is resumed from the previously hang point at the destination host [11].

Generally speaking, because the approach is realized at the JVM level, much of space and time overhead caused by source-code preprocessor and byte-code instrumentation solution at application level can be eliminated. However, the approach is not portable because it is tightly coupled with only some specific types of JVM. Unfortunately, in a applied scenario of migrating OSGi component in a heterogeneous mobile network, it is likely impossible to apply any approach at JVM and the approach is not excluded.

3.2.1.3 Source code preprocessor

At application level, some projects show a well-formed proposition in order to neglect and overcome a misfortune of non-portability caused by the aforementioned approaches at JVM level to support strong migration. The main idea of an approach proposed in the section is demonstrated as following. A Java compiler is utilized to insert Java statements into an existed source code to save states of an execution unit to a Java backup object. The modification is only made at language level and implementation of Java Virtual Machine remains untouched. It is possible to give names of some most prominent systems which follow this approach: Wasp [7], JavaGo [6] or a strong migration agents system built by three Chinese researchers[22].

A motivation behind the approach is to not modify Java Virtual Machine. Regarding to this approach, source code of a program is transformed by an external preprocessor which extends source code of the original program with some added statements. And a purpose of this action is to save values of all local variables of methods and a program counter into a backup object [5]. When the migrated Java object requires a snapshot of its captured states, it just extract them from the backup object. And a restoration phase is accomplished by re-executing a different version of original Java object's code generated also by the external preprocessor. After that, the stack are reconstructed and all local variables are initialized with values supplied by the backup object [22].
The instrumentation can be done by parsing source code of the primary program using a Java parser going along with the JavaCC, the Java compiler tool. The preprocessor then exploits and changes the parsed tree from which the new code is generated [22].

Adding new statements to the original program's source code addresses two main targets:

1. The first goal to create a reserved object containing a Java object's states. The backup object is stick together with every single Java thread that is executed inside a Java mobile program. Consequently, states of these threads comprising values of all member class variables, values of all local variables inside each method local stack and current value of program counter can be put into the backup object. The backup object has got its own Serializable format, hence it can be transferred over network with the original Java application in an uncomplicated way [5].

2. The second goal of adding new statements by the preprocessor is that it does an actual capture execution states and re-establishment the states when a mobile program is restarted at a target machine. By the way, any arbitrary execution point of the original application can be touched in the backup object. Afterwards, the restoration phase is accomplished by re-performing the modified version of the original application generated by the preprocessor right after the Java mobile object reaches the target host. The reproduction phase can recreate again a Java stack with proper frames as its original one [5].

Intuitively, it is possible to see a lot of limitations brought by this approach. First, instrumenting and inserting source code with new statements introduces time and space overhead since many additional codes are added to source code of the original program. Certainly, there is always a file size penalty because a number of lines of code are blown up significantly. At runtime point, there is also a memory space punishment since values of all local variables of methods are captured and stored somewhere in an internal memory. Furthermore, it also endures from increasing execution time at runtime point dramatically because an additional amount of time is necessary to gather up the program states after the state saving process is put into operation. Furthermore, a considerable period of time also is needed to re-establish the program states before a normal execution of the program continues. Second, the preprocessor requires all methods that might initiate a states saving process are instrumented [5].

Normally, this approach requires availability of the program's source code. An OSGi bundle often comes without its source code. Hence, it is slightly impossible to use this approach if only program libraries and binary OSGi bundles are available. Furthermore, one of its downsides is that not all of Java Virtual Machine instructions are available at the source code level, then the solution is incomplete.

And even though an assumption of availability of a program's source code can be made, many other stimulating challenges are necessitated to be cope with. First, a mobile program must be written in an extended Java language with JavaGo [6]. Then JavaGo [6] uses its own compiler to translate source code of the original mobile program into the standard Java language [6]. In a generic circumstance, it is not permissible to use any Java compiler except the standard one. Second, an IP address of a destination host must be pre-defined at a compilation time. Thus, source code recompilation must be carried out if there is a need to change an IP address of the target host.
Clearly, it is unbearable to stop and re-run a JVM on which the OSGi Platform is placed on top because it breaks the everlasting running nature of OSGi. Additionally, OSGi component might comprise many services and each service runs in its own thread. By the way, an OSGi component then can be regarded as a normal Java multi-threads application. From views of other services, a routine to capture states for each single service must occur at every execution instruction. But it is completely costly and wasteful of time to save the service’s states after each instruction executed [22]. Therefore, authors of the paper [22] came up with a solution by using a locking mechanism. The locking mechanism can be stated in the next sentences. A thread only performs its states saving routine if all other running threads released already their requests to save states, otherwise the thread blocks itself until it acquires an appropriate lock to carry out the states saving action. Nevertheless, capturing the states and re-establishing a thread's execution states using locking mechanism may encounter with the complex problem of deadlock caused by a trouble of multi-threads concurrency [28].

Source code preprocessor transforms original source code of the Java program at compilation time, hence it is not flexible and clearly it does not provide a transparent strong migration. Source code preprocessor also does not provide any ability to access entire states of a thread because some parts of the state is internal in JVM. For example, it is not possible to do a migration during an execution of a finally clause. The reason is that this clause requires the manipulation of a returnAddress value, a memory address, which can not be transported from one machine to another [5]. The approach therefore is incomplete.

From many analysed reasons and encountered difficulties, it is insufficient and really hard to employ the source-code preprocessor mechanism in runtime OSGi environments to acquire strong migration of OSGi bundles.

### 3.2.1.4 Strong migration based on byte code transformation

Another approach for supporting migration capability is post-processing an application's Java classes after it is compiled successfully. JavaGoX [3], Brakes [4] and MAG [25] employ this approach to exploit a manipulation of a specified program's byte code. This approach is similar to the source code preprocessor approach, except it works at the lower level and after the compilation phase. Instead of modifying the application's source code, the application's byte-code is transformed to append code blocks in order to catch and restore execution states.

This byte-code instrumentation procedure is performed at the Java classes loading-time by a custom class loader. Normally, byte-code instrumentation is performed by using the Byte Code Engineering Library (BCEL) or ASM, the alternative for BCEL. BCEL provides an easy way to use API for static analysis, dynamic creation and transformation of Java classes. It enables developers to implement the desired features on a high level of abstraction without handling all internal details of the Java classes files format. ASM is a Java byte code manipulation framework. It can be used to dynamically generate stub classes or other proxy classes, directly in binary form, or to dynamically modify classes at load time, i.e., just before they are loaded into the Java Virtual Machine. ASM furnishes same functionalities as BCEL but it is much smaller and faster than BCEL.

JavaGoX [3] and Brakes [4] use static byte-code instrumentation. Authors of the paper work
through doing experiments with the byte-code instrumentation approach claimed that approximately 50% of additional space overhead can be observed while running a simple recursive Fibonacci method. Therefore, they stated that a price paid for a considerable amount of supplemented byte codes placed in all Java classes files is really expensive concerning to a large space overhead. A byte-code transformation technique stated in the paper [26] is a proper candidate that fulfils all three requirements:

1. Existing Java applications can be integrated with additional functionalities in a transparent manner [26].
2. Portability of executions of distributed threads across distinctive heterogeneous platforms [26].
3. The mechanism of distributed threads can be dynamically integrable with the existing middle-ware platforms such as Java RMI, OMG CORBA, and Microsoft DCOM [26].

This approach is rather practical than the proposal of employing source code preprocessor but it suffers from space and time overhead as well. However, with byte-code instrumentation technique, a custom class loader can only automatically perform a routine to transform byte code of a particular mobile program at load-time. It is insufficient to deal with a problem of migrating arbitrary runtime OSGi bundles. Because runtime OSGi bundles must be installed and started to run by the OSGi class loader but not any other class loader. Thus, instead of instrumenting byte code at load-time, it is necessary to interweave byte code of a compiled OSGi bundle at runtime. Therefore, the existing byte-code instrumentation approach somehow must be extended in order to permit a routine of interweaving byte-code at runtime.

In the paper named “Object and Process Migration in .NET”[27], writers propose a promising solution to support strong migration of objects in .Net framework on a basis of AOP (Aspect Oriented Programming). The goal of the project is obtained by a way of extending migrants codes with some aspect codes created by mechanisms of AOP. The main idea behind the scene is that all migrants, or classes that are serialized, must be indicated with a Serializable attribute. But program counter is still missing and it can not be saved. Therefore, they mark a particular re-entry-method within the migration aspect code, which is called automatically after restoration of the migrants at the destination host. And so, they have solved out the problem because instruction pointer for the current point of execution is not saved by the serialization mechanism. However, the serialization is in a class scope, therefore all relevant state information must be enclosed in a form of class members. Data outside a migrant's scope is not included by the mechanisms of serialization [27]. Intuitively, it is likely sure to realize what the writers proposed is not completely strong migration because some execution states information is still missing: values of local variables inside each method call, frames stack.

From [27], what can be learned and can be applied to support strong migration of OSGi components is an approach of employing AOP. Obviously, it is vital to provide dynamic byte-code instrumentation in case of supporting strong migration of OSGi components. An usage of DynAOP [19] or Nanning [20] might be the alternative and feasible options. So generally, from a point of view of AOP programmers, a migration process can be illustrated out in the following paragraph. Since an OSGi component can be considered as a black box because it is hard to look insight its
structure and its code segments. Therefore, all methods of the mobile program must be traced together with appropriately passed arguments. The tracing procedure can be done by cross-cutting every single method of the Java program. It certainly incurs high overhead concerning space and time. In addition, it is necessary to do a static code analysis with the program to determine which fields which should be traced. As far as it can be known, by using AOP techniques, it is likely able to cross-cut all local fields of a specified Java class. It would be allowed to replace these fields by particular instances of objects which wrap the original fields contents and map them to a global registry, or a cover-all backup object, where values of all relevant fields can be kept. The global registry is used to store all collected results of the tracing routine with all methods. An assumption about a sufficiently big storage must be made because the system can run on resource-constrained devices like PDA, Palm Top, etc. Regarding to restoration phase of saved states, it is straightforward to perform cross-cutting all local fields once again to assign their values with the corresponding values stored in the global registry. Besides that it must be able to replay the tracing procedure by invoking corresponding methods in exactly the same order and with the same passed arguments values as they are captured previously. Note that the proposal might only work out if source code of the program is deterministic, otherwise the replaying process might end up with different results in comparison with a result created by the original one. Furthermore, it has to be assured that the migration routine has to preserve atomicity of all methods. It means that a process appending code blocks in order to capture and re-establish execution states of any mobile program should not happen in the middle of any single method call. There is still something missing, namely program counter, frames stack, etc. The information is resided internally in JVM. Therefore, it should likely be possible to inject some captured captures into JVM. A modification of JVM is likely unavoidable.

In general, similarly to the source-code preprocessor, the proposition of utilizing byte-code transformation also bears the same problem of time and space overhead. However in comparison with source-code preprocessor, it endures less overhead and it provides accessibility to an extended set of instructions at byte-code level. In addition, it is also truly challenging to trade with a difficult problem of capturing and re-establishing relevant states information of multi-threading programs. Truthfully in a generic case, an OSGi component consisting of multiple services is considered as a normal multi-threading program. Therefore, the byte-code instrumentation approach can only be applied to cope with a trouble to support strong migration of OSGi component if much more efforts in intensive researching and studying as well as doing plenty of practical experiments must be donated. Note that byte-code instrumentation approach is not able to access entire state of a thread due to a reason that some portion of the relevant state information are kept inside JVM. The approach then is also incomplete.

3.2.1.5 Java Platform Debugging Architecture modification

Standard Java Virtual Machine does not expose any feature to capture and restore the execution states of running threads. Other existing migration solutions either rely on modified standard Java Virtual Machines or instrumenting a Java program either at source code or byte code level in order to capture and restore its runtime execution states. These approaches may work well but they lead to a lack of portability (e.g. modification of JVM) or they incur a very huge
performance overhead due to injected codes for capturing and restoring states (e.g. source code preprocessor and byte code instrumentation). In DGET [10], authors choose Java Platform Debugger Architecture to perform these similar tasks. Java Platform Debugging Architecture, or JPDA for a short-hand, is a part of JVM specification and it is implemented in most of standard JVM implementations. JPDA provides accessibility to runtime information inside JVM including stack frames, local variables and program counter[10].

JPDA is implemented purely in Java, so migration solution does not destruct portability. However, this approach is still challenged from performance overhead and incompleteness of states capturing. Moreover, a remarkable problem on this proposal is a problem of re-establishing a program counter. Additionally, the JVM must be launched in debug mode [10]. Actually, it is impossible to run JVM in debug mode in many real life systems. It definitely prevents any system from applying this proposal to realize a concept to migrate an OSGi bundle between two different OSGi Service Platform running on two Java Virtual Machines in the normal mode. Therefore, this approach is not a possible solution.

Moreover, the approach also forces the Java JIT compilation to be disabled. Hence, a enormous overhead that is forced upon the mobile program executed is also a serious problem of this proposition. Some additional drawbacks are that, regarding to the incompleteness of the solution, the JVMDI does not provide any way to access the operand stack items. So that this approach is charged with a couple of limitations on using of expressions at the application programming level [10]. However, JVMDI needs big data structures and it incurs large time overhead in supporting general debugging functions. Moreover, the JVMDI-based approach needs to have Java applications code compiled with debugging information using specific Java compilers such as the javac in Sun JDK. By a mean of the reason, it will deny many Java mobile programs distributed in byte-code format without debugging information to be migrated properly. Furthermore, not all existing JVMs have realized the JVMDI defined in Sun JDK [23].

3.2.1.6 Continuation Concept and Javaflow

Jose A. Ortega-Ruiz, Torsten Curdt, and Joan Ametller-Esquerra [21] present a new approach to provide strong migration of mobile agents based on Continuation concept. Continuation is a powerful concept from the world of functional languages, like Scheme, but they are becoming popular in other languages as well. Additionally, Continuation can be regarded as a single functional object comprising execution states of a computation unit. Continuation is exactly an object which, for a given point in a specific program, consists of a snapshot of its Java stack's trace, including all local variables, and a program counter. It is capable to not only store these things in the Continuation object but also restore an execution states of a program from a Continuation object[21]. Authors of this paper also claimed that, each resume and each suspend cycle in the computation unit can be easily conceptualized as a Serializable Continuation. If a programming language allows programmer to manipulate explicitly the Continuation object then the task to support the strong migration concept can be simplified. Unfortunately, Java is a language with a poor support for explicit Continuation object handling. Moreover, current implementations of JVMs have no support for Continuation concept [21]. Hereby, they present a new library, Javaflow, which extends a standard JVM with a thread serialization mechanism and an explicit
Continuation support. Javaflow is basically developed on a basis of the disconnected project Brakes [3]. The project had already provided an implementation for Java byte-code rewriting approach. As it is already alluded previously, the basis idea of byte-code rewriting approach is to instrument Java classes with new execution instructions providing a higher control of the execution flow [21].

The crucial mechanism of Javaflow can be shortly explained as following. There is the Stack object in the package Org.Apache.Commons.Javaflow.Bytecode which acts as a virtual stack in a memory, over which Javaflow has a full control. Thus, it is likely possible to replay the virtual stack whenever it is needed. With Javaflow, there are two ways to instrument byte-code. One way is to do it statically. The Javaflow Ant-task is used as a part of a build process to enhance byte code of Java classes that run inside Continuation-enabled environment. Since the byte-code enhancement by using the Ant-task increases the class file size and slow down the execution, the alternative using Javaflow's ContinuationClassLoader can be thought over. ContinuationClassLoader works similarly to URLClassLoader except its capability to instrument byte-code. There is a need to separate a primary program into two parts: one for Java classes that do not need the byte code enhancement and the other part needs the enhancement. By this means, it can be possible to configure the first portion to be loaded by a system class loader, and the second portion is loaded by the ContinuationClassLoader. The approach sounds promising. Nevertheless, a problem of this approach can also be exemplified. In OSGi Service Platform, its class loader plays a vital role and it can not be easily replaced by any different class loader mechanism. As a part of the thesis, a couple of experiments with Javaflow have been implemented in order to investigate possibilities of integrating Javaflow into the OSGi environment. Advices from some OSGi evangelists and inventors of Javaflow have been also taken into account. However, a controversial question whether ContinuationClassLoader can live peacefully with the standard class loader of OSGi is still not answered so far. Finally, it is not completely concluded whether or not an idea to integrate concept and implementation of Javaflow into OSGi Service Platform might provide strong migration of OSGi component in near future.

3.2.1.7 Translating Strong Mobility to Weak Mobility

In the paper [29], an approach to support strong migration in Java by translating strongly mobile code into weak mobile code is presented. Authors of this paper aim to utilize the IBM Aglets weak mobility system. The fundamental idea is explained in the coming section. Every method in an original program is translated to a Serializable inner class which acts as an activation record for this method. Local variables, parameters and program counter of the method are converted to fields of the inner class. The inner class consists of run() method that replaces a functionality provided by the original methods [29].

The produced weakly mobile class comprises an array of activation record objects that play a role as a virtual method table. In the concept of strong migration, execution states of a Java thread need to be preserved so that the thread can be recommenced at the a destination host. The authors propose a way to encapsulate each Java thread with the Serializable wrapper. The Serializable wrapper has its own stack of activation records that describe runtime stacks of the corresponding thread. Therefore when any method is invoked, a proper entry from the virtual
method table is cloned and put on the stack. Its `run()` method afterwards executed. Additionally, source code of the original program is changed to a format that permits execution states of the primary program to be captured for each performed execution statement while sustaining semantics of the execution statements. Hence, execution of statements can be suspended at arbitrary point of time and that allows the original program can be migrated along with a saved program counter and it can be restarted from the postponed point. The migrated program carries along with the `Serializable` wrappers of its all threads. At the destination host, these wrappers produce again new `Thread` objects and recreate their execution states [29].

This approach is in fact a variant of the source-code preprocessor. Therefore, it endures all problems that source-code preprocessor solution suffers from, namely a major issue to preserve the atomicity of a logical instruction or a challenge to prevent deadlock when multiple threads attempt to dispatch the mobile program, etc. Moreover, the approach also uses its own Java compiler to transform a strongly mobile program to a weakly mobile program and especially the preprocessor only can generate Java code that consumes IBM's Aglets library. Hence, the approach is therefore not flexible. Similarly to solution provided by source-code preprocessor, the approach also needs to make a deal with a problem of optimizing blow-up source code. The authors claimed in the paper that many portions of their translator have not been implemented yet and the translator is a specific preprocessor to the Brew compiler. Up to the time when the paper was published, they are only able to translate not the entire Java language but some Java structures and instructions are not translated successfully. Then their solution is certainly incomplete. With many difficulties and restrictions already analysed in the section of source-code preprocessor plus its specific major issues, it is completely hard to employ the approach to realize strong migration concept of OSGi components.

### 3.2.2 Weak Migration

So far, weak migration can be implemented with reasonable efforts to be contributed. Many projects support mobile agents paradigm on above of well-known mobile agents infrastructures. Voyager [46] has a facility for mobile agents. A voyager daemon waits on a predetermined port for a mobile agent. When one arrives it unpacks it and calls a method that has been specified prior to migration. IBM's Aglets [46] and JADE [47] provide a similar agent mechanism. Some other projects [2] employ Java RMI to implement of the mobile code concept. According to the scope of the project, it is really feasible to support the migration concept of OSGi components in weak mobility manner with much less efforts.

### 3.3 Size of data transfer

Generally, strong mobility requires bigger size of data to be transferred because it enables migration of code, data as well as execution states of an execution unit, for instance a Java thread. Regarding to weak migration, which migrates only code and data. A weakly mobile system gives programmers more active control over amount of states that have to be transferred. Conversely, a strongly mobile system may bring unnecessary states, so it is implied to increase size of the
serialized data. A created backup object obtained from the source-code preprocessor and the byte-code instrumentation approaches that is used to store all relevant states information: global variables, local variables of each method call, frame stacks, operand stack, object heap, program counter have huge sizes. In order to build such a data structure, the Java stack associated with each individual thread is scanned through to recognize its current frame stacks, local variables, operand stack, program counter. After thread is serialized, the achieved data structure can be transferred from a source machine to a destination machine. And size of the the data structure normally is relatively large. Concerning to two particular source-code preprocessor and byte-code instrumentation approaches, many lines of code are added to an original mobile program. Therefore, number of lines of code are blow up significantly. This directly infers that size of classes and size of source files to be transferred are also increased dramatically. Respectively, source-code as well as byte-code in weak migration are normally not modified and added. Hence, in comparison with strong migration, size of code, data and relevant states information to be transmitted in case of weak mobility are significantly smaller.

### 3.4 Latency of migration

Strong migration and weak migration will be compared in this section on a basis of latency of migration caused by each concept.

#### 3.4.1 Strong migration

A latency of migration is defined as a summed time interval used for purposes of capturing Java thread's states, transferring and restoring relevant state information. Intuitively the latency of migration depends on a particular application which involves in the migration process. Particularly, it depends on a number of Java frames on the program's stack. In other words, a period of time used in capturing and restoring a mobile program states is dependent upon size of the relevant state information at the capturing time [5, 30]. Generally, once the mobile program is suspended, all methods will return immediately after capturing the frame information. When resuming the suspended program, it is impossible to reconstruct the frame stack in a direct manner and spring suddenly to the position where execution was left off previously. Therefore, the mobile program has to hurriedly re-operate through the complete method call hierarchy to come back to the desired position during the resume phase. Since most of already executed code in the past can be omitted, executing invoked instructions will construct required frames stack really quickly and the original states are brought back once the marked program position is arrived thanks to a process of loading frames stack from a backup object. On the other side, the transfer of a program states to a target host consists of following steps: serializing the backup object keeping all relevant state information to an array of bytes, transmitting the obtained array of bytes over the network, and de-serializing the array of bytes on the destination host. By doing some experiments with an approach at JVM-level, researchers have claimed that a cost of migrating a mobile thread depends mostly on time spent to send the backup object to the destination host. More precisely, the state transfer represents almost 98% of the total latency of a whole migration process [30]. Moreover, latency of migration with approach at application level, namely source-code
preprocessor and byte-code instrumentation, is even smaller than approaches at JVM level [5].

### 3.4.2 Weak migration

Regarding to weak migration, a mobile program must be changed from the active state to the suspended state. A mobile program is in this case in not pre-emptive. Hence there is no interruption inside every method call. We must wait for the mobile program finishing all current method calls and the migration process then is actually activated. Time used for saving and re-establishing session data of a mobile program is relatively insignificant and in fact, the period of time is strictly dependent up on the specific state capturing algorithm. Similarly to strong migration, latency of weak migration mostly depends on time spent for transferring states. The state transfer time can partly be reduced by using Java externalization rather than serialization. Externalization allows application programmers to write their own object transfer policy by saving only information necessary for rebuilding object graphs. Externalization may be until 40% faster than serialization [32]. The total latency in weak migration finally is a sum of time interval spent to wait for all current methods to be accomplished, time interval to save and re-instantiate saved session data and transfer state. To conclude, difference in latency of migration in weak mobility and strong mobility manner is insignificant.

### 3.5 Performance

Regarding to strong migration, based on the achieved results provided by [5], almost all existing solutions, at both JVM and application level, compel an considerably significant performance overhead on execution. In fact, the JVM-level approaches suffer from inducing a significant overhead on execution performance, approximately +335% to +340% [5] because they require the Java interpreter to be extended and they normally do not support Java execution optimization (JIT compilation). Due to the statements added to original code at source-code or byte-code level, the application-level approaches impose a bit less performance overhead than at JVM-level systems. However, the performance overhead at runtime is still non negligible. It fluctuates within about +88 to +250% [5]. On the other point of view, the weak migration does not stand for any performance overhead.

### 3.6 Conclusion

From previous parts, it is possible to observe and realize the mainly principal drawbacks of the strong migration implementations. It is impossible to only contribute reasonable efforts within a limited time frame either at application level and Java Virtual Machine level to solve out completely all problems derived from a concept of strong migration. And it is said that many researchers faced with these difficulties when they actualized their researching projects. Therefore, many aforementioned research-oriented activities to furnish strong migration concept have stopped some years ago. Moreover, any proposed solution for a specific platform is more or less so invasive into a JVM and consequently that can not be run in a standard JVM. And with all previously mentioned approaches, all authors assumed that there is no connection or dependency among migrative Java objects. However, OSGi bundles are naturally highly interconnected with each other.
In addition, regarding to OSGi specification, the graph of interconnected bundles can be quite huge because the Service Registry layer provides a comprehensive model to share objects between bundles. Therefore, in order to provide strong migration for a specific OSGi bundle, it must not only to capture execution states of this bundle but also execution states of all other OSGi bundles supplying services that are consumed by this bundle. And in fact, it is nearly impossible to realize this idea. So far as my knowledge, there is no existing approach to deal with such the complicated issue.

In the chapter 6 of the paper, I will describe a system architecture that provides weak migration for OSGi bundle based on Java RMI. The middle-ware can provide us capabilities for stateful cloning and relocation of OSGi bundles. I will give out a short explanation regarding a concept of term bundle state in this chapter. Migration of bundle can be in form of passive manner. In a passive manner, bundle has a capability of being cloned, where it can copy itself along with its states to another suitable hosting environment transparent to the consumer of that service.
Chapter 4 Related works

ASK-IT[12] proposes an approach to integrate a compatible agent platform JADE-LEAP framework [13] with OSGi technology. They come up with an idea to integrate JADE with OSGi because of the following possibilities offered by the OSGi Service Platform. This approach is absolutely promising but unfortunately JADE indeed only supports weak migration. So that, despite plenty of interesting services and tools provided by the system, it is rather difficulty to be succeeded from research activities to real world applications because programmers who follow this approach might be lead to design and develop applications on a basis of a mobile agent-based programming model which is far different from a daily practical object-oriented model. And especially, the system can only be implemented in the unnatural framework because system developers must follow programming principles to elaborate mobile agents system [12].

In the paper [18], authors propose an approach to set up Java-based mobile agent platform based on a combination of OSGi and Beanome. In spite of many good features of the lightweight component model and framework on top of OSGi, Beanome does not address various issues like remote communication, dynamic binding, remote invocation, etc. Therefore, authors extend the fundamental architecture of Beanome by adding Migration Manager component with purpose of supporting OSGi bundles migration. Migration Manager in turn utilizes HTTP Post request to send and receives OSGi bundles between different OSGi platform. This approach use standard HTTP and Servlet service of the OSGi specification, then it is really flexible. However, HTTP is an application-level powerless, simple text-based protocol. The main problem of HTTP is that HTTP can only communicate with Web server, hence it introduces problems with performance issues. By using HTTP Post, communication between two endpoints can especially go across a restriction of Firewall. An advanced feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred. Nevertheless, in spite of its many advantages, usage of HTTP Post requires an explicit mapping mechanism between encoded data embedded in HTTP Post message in both client and server side and in some typical case, it can cause a big complexity. Moreover, using Servlets for communication, the connection can not be kept opened continuously, since it is piggybacking on the stateless HTTP protocol. Thus, a new connection must be initiated for each request in all of methods, consequently it is a bit expensive. The paper's approach depends on Mobile Agents model, therefore it suffers from the already specified drawbacks in [12].

A research group working at IBM Software Labs in India implemented IBM Service Management Framework™ for Relocatable Services (ReSMF) [20] that provides a convenient environment for hosting and managing relocatable OSGi services. They support concept of migration OSGi bundles basically on a basic of Mobile Objects paradigm, hence programmers can get benefit of an easy and comprehensive model for programming any new relocatable service. The framework is built over the IBM Service Management Framework(SMF), the implementation of the OSGi Service Platform specification invented by IBM. The main limitation of this approach is that it is tightly bound to SMF, therefore it is not trivial to use the framework on top of other
implementations of OSGi specification like Knopflerfish, Oscar, etc. Similar to the paper [18], communication engine that allows establishment of a connection link between two different hosts is built on Servlets binding technology. Thus, it is also suffered from the already cited problems of the work [18].

Seung Keun Lee, Jeong Hyun Lee [14] also deal with a challenge to design and develop an OSGi-based mobile agent management system for maintaining a mobile agent bundle in the OSGi Framework. This paper introduces a methodology that can administer bundles like dynamic agents, in order to ensure the mobile ability of those entities among multiple OSGi Platforms. An overall mechanism of this approach is depicted like that. After receiving a mobility request, an agent is switched to a mobility request state and an execution suspension is requested. The agent receiving the request returns after completing all current actions or methods call in the main process. Status information, including object code segments and its states, prior to the mobility is marshalled into XML format(SOAP) by some other functional bundle. And the SOAP messages are delivered to a destination platform. At the destination framework, the original object is de-serialized from the SOAP messages and its former states are also restored. Unfortunately, it is not easy for readers to learn how they can migrate a bundle's states. They have implemented an experiment with a MP3 player. They claimed that they can migrate the player service and preserve the current states. However, it is likely believable that a currently playing MP3 file and value of an offset inside the file are actually migrated. SOAP has a verbose format and it is really challenging to map random Java data types into its corresponding data representation in XML format and conversely. So a concept of states saving and capturing in this paper is not extensive.
Chapter 5 OSGi-basics and implementation

In this chapter, all related stuffs to the OSGi Technology, namely the OSGi Service Framework and its standard services, clarification about a concept of an OSGi component, some open-source and commercial implementations of the OSGi specification will be introduced.

5.1 OSGi Technology

Overview about the OSGi Service Framework architecture, functionalities and APIs of its standard services like Configuration Admin, HTTP Service, Service Tracker, Package Admin Service will be sketched out in details in this sub chapter.

5.1.1 Overview about OSGi Service Framework

OSGi (Open Service Gateway Initiative) service platform, contributed by OSGi Alliance, specifies a standard environment which allows multiple, Java-based components, called bundles, to run in a single Java Virtual Machine (JVM) securely [36].

Execution environment - JVM

Java environment is selected by the OSGi alliance as the OSGi execution environment. J2SE and J2ME, for example, are valid execution environment [36].
**OSGi framework**

The OSGi framework is the spirit of the OSGi service platform, which addresses issues caused by running multiple applications in a single JVM. Its responsibilities include: class loading, life cycle management, service registration, and security issue.

1. **Class loading:** Since multiple bundles are allowed to run on the OSGi platform while the execute part are classes which are grouped in a package. How to manage these classes within bundles? OSGi accepts the solution to share classes among bundles instead of making them private for the key advantages of lessening bundle size and memory footprint. Class loading takes charge of class sharing issue [35].

2. **Life cycle management:** This component ensures bundles to be dynamically installed, started, stopped, updated and uninstalled [35].

3. **Service registry:** As its name indicated, service registry is the container for services. Services are just Java objects published by one bundle so that other bundles could use them, while Java objects could be anything, HTTP server service, for instance, is a Java object. Java object is registered with an interface name and a set of properties [35]. For example, the OSGi HTTP Service would be registered with the `org.osgi.service.http.HttpService` interface name and properties such as `vendor=Knopflerfish`. Service registry enables bundles register their objects, looking for matching services and being notified when services coming or going. Service registry therefore links bundles together and turns the OSGi service platform into a component framework [35].

4. **Security:** OSGi security is based on Java 2 security model. The access modifier in Java 2 security model makes classes, methods, and fields private (accessible only by classes in the same package), protected (accessible by sub-classes) or public. OSGi extends this model by adding an extra level `package private` to the bundle which indicates these packages are only accessible by code inside the package but restricted by other bundles [35].

**OSGi Bundles**

Bundles are Java applications packaged in a standard Java Archive (JAR) file. They could be dynamically installed, started, stopped, updated and uninstalled by the OSGi framework. There exist two kinds of bundles:

1. **Standard Bundles** that implement OSGi standard services. OSGi specifies a set of standard services in an abstract level and could be optionally selected to be implemented in different ways by different vendors [35]. For example, both organizations, Knopflerfish and Oscar, implement OSGi framework as well as a set of standard services, but Knopflerfish makes OSGi framework to be a GUI while Oscar a console.

2. **Customer Bundles** that are provided by third parties. These bundles implement customer-specific functionalities. Normally, customer bundles make use of services provided by standard bundles [35].

**5.1.2 Configuration Admin Service**

The Configuration Admin service is an important facet of the deployment in the OSGi
service platform. It permits configuration of deployed bundles can be set by system operators. In other words, main intention of this service is to store bundle configuration data in a persistent manner. This configuration data is represented in Configuration objects. The factual configuration data is a Dictionary of properties inside a Configuration object. A bundle can receive Configuration objects by registering a configuration service with a PID property. In principal, there are two distinct ways to administrate and handle configuration data. First there is a concept of the Managed Service, where configuration data is connected with an object registered with the Service Registry in the unique manner. Second there is a concept of Managed Service Factory. Configuration Admin service will provide 0 or more Configuration objects for a Managed Service Factory that is registered with the OSGi Framework [35].

When the Configuration Admin discovers a registration of the Managed Service, it checks its stubborn storage to find a configuration object whose PID is equal to the PID of the Managed Service or Managed Service Factory. It will can updated() method of the Managed Service class with the new properties in Dictionary type if the matching is found. The implementation of Configuration Admin service must perform a callback in an asynchronous way to allow proper synchronization. Similarly, when Configuration Admin detects registration of the Managed Service Factory, it all look for configuration objects whose factoryPID matches factoryID of the Managed Service Factory. With each found Configuration object, the updated() method of Managed Service Factory class will be called asynchronously with new properties [35].

When a Configuration object is created by either getConfiguration() or createFactoryConfiguration(), it becomes bound to location of the calling bundle. This location is obtained with the associated bundle’s getLocation() method. A null location parameter may be used to create Configuration objects that are not bound. In this case, the Configuration Admin service will bind it to location of the bundle that registers the first Managed Service or Managed Service Factory that has a corresponding PID property [35].

5.1.3 HTTP Service

The Http Service allows other bundles in the OSGi environment to dynamically register resources and Servlets into the URI Namespace of Http Service[35]. In the other words, the Http Service is, among other things, a Servlet runner. Bundles can provide Servlets which becomes available over the Http protocol. The dynamic update facility of the OSGi Service Platform makes the Http Service to be a very attractive web server that can be updated with new Servlets, remotely if necessary, without requiring a restart [37]. More late, bundles can unregister resources and Servlets. OSGi specification defines HTTP Context and HTTP Service interfaces. HTTP Context enables bundles to provide information for a Servlet or resource registration. Servlet objects can be registered with the Http Service by using the Http Service interface. For this purpose, the Http Service interface defines the method registerServlet(String, javax.servlet.Servlet, Dictionary, HttpContext) [35].

5.1.4 Service Tracker

The OSGi Framework provides a powerful and very dynamic programming environment.
Bundles are installed, started, stopped, updated, and uninstalled without turning off the Framework. Dependencies between bundles are observed and kept track of by the OSGi Framework. However, bundles themselves must work together in order to handle these dependencies correctly. For instance, bundle must not use the service objectives that are already unregistered. That is a potential problem. Fortunately, the OSGi specification provides a utility class, Service Tracker, that eases tracking the registration, modification, and unregistration of services. Therefore, with existence of Service Tracker utility class, the complexity of tracking services in the Service Registry is reduced significantly [35].

A Service Tracker has some following fundamental tasks. First, it is used to create an initial list of services that are specified by its creator. Second, it listens to Service Event instances so that services of interest to the owner of the Service Tracker are properly tracked. Third, it allows the owner to personalize its tracking process through programmatic selection of the services to be tracked as well as when service is added or removed [35].

A Service Tracker class can be fitted according to individual needs or desires by providing a Service Tracker Customizer object implementing the desired behaviour when the Service Tracker object is created, or by sub-classing the Service Tracker class and overriding Service Tracker Customizer methods: addingService(), modifiedService(), and removedService(). These three methods are called respectively when a service of interest is being added to the Service Tracker, or when it is modified or removed from the Service Tracker object [35].

5.1.5 Package Admin Service

In OSGi Service Framework, bundles can export packages to other bundles and this exporting and using creates dependencies between these bundles. OSGi provides Package Admin service. It provides access to internal structures of the Framework related to the packages sharing. Package Admin provides the following methods, getExportedPackage(String) and getExportedPackage(Bundle) and refreshPackages(Bundle[]). The method getExportedPackage(String) returns an Exported Package object that provides information about the requested package. The method getExportedPackage(Bundle) returns a list of Exported Package objects for each package that the given bundle exports. The method refreshPackages(Bundle[]) allows to refresh the exported packages of the specified bundles. Information about the shared packages is supplied by the Exported Package objects. All detailed information about the bundles that import and export the package are provided by these Exported Package objects [38].

5.2 Definition of OSGi component

OSGi component can be called as OSGi bundle. An OSGi bundle acts as a representative on behalf of a JAR file that is executed in an OSGi Framework. The OSGi bundle can be installed by another bundle and the bundle is started to execute its functionalities through its Bundle Activator. The manifest header file of the OSGi bundle identify its Bundle Activator. And the given class must implement the default Bundle Activator interface. The Bundle Activator interface has two default
methods: `start()` and `stop()`. The method `start()` is invoked when a bundle programmer wants to register it as a listener and start any necessary threads. The method `stop()` is called with a purpose to clean up and stop any running threads [38]. Concretely, a bundle is a Java JAR file that contains a manifest and some combination of Java class files, native code, and associated resources. The manifest of the bundle JAR file contains meta-data describing, among other things, the Java packages that the bundle requires or provides.

There is an associated Bundle object for each bundle installed in the OSGi Service Framework. The life cycle of the bundle can be managed through using the corresponding Bundle object. The bundle can be identified uniquely by Bundle Identifier, Bundle Location or Bundle Symbolic Name. Bundle Symbolic Name is a name specified by the bundle developer. The combination of Bundle Version and Bundle Symbolic name is globally unique identifier for a bundle. Especially, in the implemented system in the scope of the project, Bundle Symbolic Name is really useful to prevent two duplicated installations of one specific bundle in a single OSGi Framework [38].

When a bundle is installed, it can have one of the following states: INSTALLED, RESOLVED, STARTING, ACTIVE, STOPPING [38].

![Figure 4: Bundle states diagram](image)

Bundle Context interface provides two methods for installing a bundle, `installBundle(String)` and `installBundle(String, InputStream)`. The first method with one input parameter installs a bundle from a specified location, namely URL. The second one with two input parameters installs a bundle from a specified Input Stream. The installation of the bundle in the OSGi Framework must be persistent and atomic. The bundle can be started via the `start()` method of Bundle interface. If the method is called successfully, the bundle state is changed to ACTIVE and it remains in the state until it is stopped. Intuitively, to be started, the bundle must be first resolved. If the bundle is resolved, the bundle must be activated by
calling its Bundle Activator [38].

In the OSGi Service Platform, a bundle comprises of a set of available cooperating services. An OSGi service is defined by its service interface, implemented as a service object. The service object is owned by, and runs within, a bundle. The bundle must register the service object with the OSGi Framework's service registry so that service's functionalities is available to other bundles. When the bundle is stopped, all the services registered with the Framework by the bundle will be also unregistered. Dependencies between bundles offering services and bundles using them is managed by the Framework. The Framework provides a query mechanism, so that a bundle can request all services it needs. The OSGi Framework also provides an event mechanism. By the way, bundles can receive events of service objects that are registered, modified, or unregistered. Registered services are referenced through Service Reference objects. A Service Reference encloses the properties and all meta information about the service object it represents in a capsule. A Service Reference object can be stored and passed to other bundles without implying any dependencies. When a bundle wants to use the service, it can be acquired by passing the Service Reference object to the method `getservice(ServiceReference)` of Bundle Context object. Hence, this avoids creating redundant dynamic service dependencies between bundles when a bundle needs to know about a service but does not require the service object itself [35].

5.3 OSGi implementation

Open Source implementations and some commercial products that implement OSGi specification will be presented to readers in the next following sections.

5.3.1 Open source implementation

Oscar is an open source implementation of the OSGi Service Framework and has been available since the year 2001. The current version of Oscar is 1.0.5 that was released on 16th of May 2005 [39]. Now Oscar is maintained and further developed by Apache Felix [42].

Knopflerfish is a name of the project from Gatespace, which was one of the founding member of OSGi Alliance. The goal with the Knopflerfish project is to develop and distribute easy to use open source code, build tools and applications, related to the OSGi framework. The current version of Knopflerfish is 2.0.1 released on the 30th of November 2006 and this version is compliant with OSGi R4 Specification [40].

Eclipse-Equinox is an implementation of the OSGi R4 Specification, a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems. The primary goal of the Equinox project is to be a first class OSGi community and foster the vision of Eclipse as a landscape of bundles [43].

Concierge is implemented by the Information and Communication Group(IKS) at Swiss Federal Institute of Technology(ETH Zurich). Concierge is an optimized OSGi R3 framework implementations with a file footprint of under 80 kBytes. This makes it ideal for mobile or embedded devices. Typically, these devices have VMs that are more focused on compactness and less optimized. For instance, purely interpreting VMs often kill the performance of existing OSGi
framework implementations. The design of Concierge has been developed with respect to such platforms. Concierge uses resources in a very careful way and is able to provide significantly better performance in resource-constrained environments [41].

### 5.3.2 Commercial implementation

The mBedded Server Professional Edition is based on the ProSyst OSGi framework implementation (OSGi R4 certified) and offers even more benefits - it contains everything that is needed to embed this platform into mass market devices. It provides highest availability, scalability and reliability. mBedded Server Professional Edition sets new standards for performance and memory efficiency through several unique technology innovations. It’s optimized for small platforms with limited resources. It comes with all warranties and liabilities rights that are necessary to ship devices to end users [44].

Gatespace Telematics maintains and supports Knopflerfish Pro, the commercial edition of the open source OSGi Knopflerfish. Knopflerfish Pro is a complete certified OSGi release 4 compliant service platform. Knopflerfish Pro is a fully supported open source based product and it is primarily intended for corporate use. Knopflerfish Pro offers all the assurance required for companies to commercially use open source software [45].

Additionally, some other commercial products like Service Management Framework (SMF) of IBM or some other commercial products developed by the Connected System Incorporation and the Atinav company.
Chapter 6  Stateful weak migration solution for OSGi component

6.1 General requirements

- The first objective which should be touched is the ability of the system to operate in a heterogeneous network.
- The second general requirement is the system performance and the system can run in not only powerful computers but also resource-constrained devices like Palm Top, Pocket PC.
- The implementation of the system should not break original implementation of standard JVM is the third prerequisite.
- Additionally, OSGi component can be relocated from one machine to another machine with its saved session states. And session states can be saved in forms of various data types from Primitive Data Types to Complex Data Types and even in User Predefined Data Types.
- Communication technologies should be transparent to application programmers who take charge of implementing migrative OSGi components. In other words, it is possible to somehow replace communication technologies but the substitution at the communication layer does not affect implementation at the application layer.
- It is also really valuable to optimize efforts of application programmers contributed to put migrative OSGi component into correct operations. Hence, they can only concentrate on business logic functionalities of migrative OSGi components.

6.2 General Design

In the aforementioned applied scenarios of the project, namely load balancing or local resource access, etc. Hence, the term “Migration” can be used for movement operation of passive objects. And the code shipping is also selected as the transfer paradigm of the system. To state the matter differently, weak migration in a form of a combination of code shipping model and moving only passive mobile objects is the method to realize the system.

The overall architecture of the elaborated system under a scope of the student project can be described roughly as following:
From a general perspective, the system can be divided into four distinct layers: the Java Virtual Machine layer, the service layer or a runtime container, the communication layer where Migration Manager component is allocated, the application or business layer where Component Manager, Migrative Component and their dependent components are implemented. Note that those dependent packages are not explicitly drawn out in the figure.

Application programmers can involve in extending functionalities of the default Component Manager but their main tasks are to implement Migrative Component.

End users like a system administrator can use the system for his own job, namely to perform a task of balancing runtime loads between many machines in a LAN network by explicitly moving some bundles from machines with heavy loads to machines with less burden.

Migration Manager is the most important component in the system. It provides a monitoring mechanism to keep track of every migrative component’s state in the system and checks these components up on if they are in the “ready to be migrated” state. After that, Migration Manager will establish a communication pipe with another Migration Manager component in a destination host in order to move the “ready to be migrated” component. Actually, IP address of the destination host must be somehow pre-defined before the migration takes place. Up on the migration process of the migrative component, its session data must be extracted in some manner and it will be transmitted along with code fragments of this component. Extracting and re-instantiating session states of a specified migrative component are the obligatory duties of Migration Manager. Obviously, the migrative component has to give Migration Manager a possibility to extract and re-establish its session states via some pre-defined interfaces.

Component Manager supplies end users with a friendly-looking interface to manage all migrative components in a local framework in such a way that they can select any particular
component to be suspended and migrated. And end users also definitely choose an IP address of a destination host. Afterwards, the migration routine will be handled automatically and transparently to end users by the Migration Manager component.

A migrant, or a Migrative Component, will be elaborated by application programmers based on some pre-defined programming interfaces included in a specific package.

6.3 Comparison RMI with Socket, Web Services and SOAP

Communications of SOAP and Web Service normally use HTTP. HTTP is universally supported and HTTP can pass easily through firewalls. RMI is the Java centric technology, whereas SOAP and Web Service use XML that is language independent. The use of HTTP and XML text documents supports increases interoperability but also represents a considerable growth in run-time cost for SOAP and Web Service solutions as compared with Java RMI. Java RMI uses optimized connection oriented communication protocols that are either language specific or have detailed rules defining how data structures and interfaces should be realized. Java RMI connection is implemented based on TCP transport protocol. And SOAP and Web Service are realized at the application-to-application level and they use textual protocols like HTTP, SMTP. The XML formatted documents are essentially more voluminous than the binary data traffic of Java RMI. Certainly, the more data have to be exchanged across the network, the more control packets are required. Hence, it is considerably costly to use SOAP and Web Service. In terms of Java-RMI, Web Service acts as a singleton server object. The singleton server character of Web Service implies that the stateless server architecture is designated. The stateless hypertext transfer protocol, HTTP, used by SOAP and Web Service was invented at first for a principal purpose of downloading individual file and it is not ideally suited for applications where multiple requests and responses maybe need to be exchanged. The logical reason is that SOAP and Web Service work on the request and response model of making remote procedure calls, and a new connection is re-established if a new remote invocation call between two end points is performed. Intuitively, this is really expensive in some typical cases. Especially in the scope of this project, Migration Manager allocated in two different hosts usually exchange large number of information in forms of multiple requests and responses in order to not only transfer the binary content of a specified migrative OSGi component and its cover-all backup object representing session states of but also plenty of its dependent OSGi packages as well [33].

The major downsides of SOAP and Web Service in comparison with Java RMI regarding to the scope of the project are raised up when there are demands to work with arrays type and complex Java types. RMI uses Java serialization mechanism to convert contents of an object into a byte stream that can be sent across network. Both client and server must hold definitions for the data types being passed. And the remote interface definition uses standard Java syntax to indicate where objects are used as parameters or return values. In contrast, when using complex Java types as part of SOAP and Web Service, it is unavoidable to address the same issues. However, there are plenty of added complications. For example, the difficult matters must be dealt with in the platform-independent and language-independent way. So that, the mechanism to marshal and unmarshal contents of the complex Java types into corresponding XML formats that can be used as parts of SOAP messages must be definitely and explicitly provided. Hereby, XML parsers that allow client
and server implementations to construct their distinct but equivalent representations of any data structures must be implemented. By the way, the marshaling and unmarshaling mechanisms can be performed on both the client and the server sides. Additionally, in case Web Service uses WSDL to describe service interfaces along with details of their bindings to specific protocols, there must be also some indications in WSDL file to reveal certain parameters or return values are complex types. It is indispensable to consider also a situation where WSDL generated from a non-Java Web Service is provided. To use the Web Service from the Java environment, WSDL file in this case is obligated to contain definitions for complex types that must be mapped into corresponding Java types. Actually, the type-mapping problem induced by SOAP and Web Service can be handled but it is not straightforward. For instance, some specifications related to SOAP and Web Service define serialization frameworks that can be used to map between complex Java types and their XML representations in WSDL and SOAP messages. In this case, Serializer and Deserializer classes for each complex type that are called by Web Service to perform the types conversion must be explicitly declared and implemented. However, concerning to the special objectives of this project, the raised issues are likely challenging to be touched with our hands because a number of services inside a specified migrative OSGi component and data types of class member variables of every Java class of each service are varied considerably. It means that it is complicated to encode and decode session states of the OSGi component if those session states are represented in XML formats. The concept of nested backup object aforementioned in the section 3.2.1.3 is consequently hard to be actualized. Moreover, application programmers have to contribute much more efforts in order to supplement for insufficiency of the mechanism to marshal and unmarshal complex data types provided by the underneath communication layer. In contrast, with Java RMI, application programmers give almost no effort at all. In fact, there are also some unexpected and unforeseen problems that we may encounter if we implement the project with SOAP and Web Service.

Fundamentally, a decision to employ SOAP and Web Service or RMI is given on the basis of used programming languages. From Java programmers point of view, RMI is the most elegant approach. If we never need anything but only Java, SOAP and Web Service loose most of its benefits. Intuitively, OSGi specification was originally invented to target the Java environment. In addition, communication technologies are solely utilized as a part of the system at the communication layer. And all components of the system do not need to interoperate with other applications or components written in any other programming language. Basically, system developers will have to choose between interoperability where Web Services have advantages, and performance that will favour to Java RMI. Under the scope of the project, interoperability is unnecessary at all but performance is vital when resource-constrained devices must be addressed [34].

The main problem of Java RMI is that RMI originally can not bypass through firewalls. However, there are still some acceptable solutions. The easiest way is that if we have proper administration rights in the whole network, we can easily open some specific ports and that can allow RMI applications to operate properly. In another way, Java RMI may be encapsulated within HTTP or routed through firewalls using protocols like SOCKS. Since SOAP overhead is greater than Java RMI even discounting HTTP overhead, it is expected that Java RMI will be faster over HTTP as well. Another solution is to use RMI application proxy. There are some commercial
products in the market, like RMI Proxy developed by Telekinesis Pty Ltd company....And the price of this product is reasonable and it is possible to afford such amount of money in a necessary case.

RMI is an abstract layer of Java Socket with many additional advanced features: object oriented model, automatic encode and decode Java data types, multi threading, exception and error handling.....Therefore, RMI is much better solution than Java Socket.

Therefore, despite of many advantages of SOAP and Web Service, the project will employ RMI because RMI is so far the most appropriate technology just for the typical objectives of the project. It is also really worthy if SOAP and Web Service can be used to make extension of the project in some aspects if there will be any vital need.

6.4 System Architecture In Details

Figure 6: The detailed System Architecture

The detailed description of the system can be depicted with the figure 6. OSGi Service Framework is selected as a runtime container for the system because with OSGi allows a bundle to be installed, started, stopped and removed without affecting operations of other bundles in the same local framework. This characteristic of OSGi highly fits with desired features of any mobile system.

The virtual communication channel that is described in the figure 5 between two Migration Manager components in two different hosts is implemented on top of RMI.

Migration Manager component can be specified as a compound of different objects, namely Migration Manager Server, Migration Manager Client, Http Stub Downloader (see technical details in the section 7.2.4) and Migrative Service Tracker.

Migration Manager Server and Migration Manager Client objects perform central roles in the system. They act as a server side object and a client side object disparately. Migration Manager Server must provide a least a couple of basic methods that are called by its corresponding object in
the client side to get information about an existence of any bundle as well as to receive and install an arbitrary bundle with or without its session states respectively. Migration Manager Client must have not less than a method that is invoked whenever any bundle need to be migrated. Detailed descriptions of Migration Manager Server and Migration Manager Client can be found in the section 7.2.2 and in the section 7.2.3.

The Migrative Services Tracker object receives a duty to listen on the Service Registry of the OSGi Service Framework to find if there is any Migrative OSGi Bundle registering its “Ready to be Migrated” state. If it is found, the Migrative Services Tracker will in turn make a call to a specific method of the Migration Manager Client object to make a connecting request to the corresponding Migration Manager Server object at the server side. A connection established between two hosts will be used to transfer the bundle along with its saved states and its relevant bundles (see details in the section 7.2.1).

Migrative Component can be called as Migrative OSGi Component and Component Manager is recognized as Bundles Manager. Details of Migrative OSGi Component and Bundles Manager can be found in the sections 7.1 and 7.3 respectively.

6.5 Interaction and co-operation between the system components

![Diagram showing interactions between system components](image-url)
The overall execution flow can be explained briefly in following sentences. An end-user defines an address of a destination host by typing arbitrary IP value manually. Concurrently or later, he might make a choice in order to select an appropriate OSGi component from a list of active bundles. Subsequently, he can make a confirmation to definitely locate the destination host. The Bundles Manager component in turn will send a request to the standard Configuration Admin service in order to identify the target host. At the later time, Migration Manager Client will get a petition from Configuration Admin for setting up a communication pipe with the recognized destination host. The reason why Bundles Manager does not send the IP value directly to Migration Manager Client but indirectly through Configuration Admin is explained in the section 7.2.3.

After successful connection between two hosts is established, Migration Manager Server will register its remote reference as an ordinary OSGi service in the client side.

Afterwards, Bundles Manager will force a migrant, or a migrative OSGi component, to be suspended and to be migrated. The Migrant changes its migration state and notify the Migration Manager Client service that it is ready to be immigrated. Upon the request, Migration Manager Client try to get all information about the migrant's dependent packages. Transferring all these packages from the source host to the destination is the next step. Migration Manager Server takes responsibility to install all these bundles in advance.

More late, Migration Manager Client give a try to collect all relevant session states of every service residing inside the Migrant and load the session data into a cover-all backup object. In the next step, Migration Manager Client makes an appropriate remote invocation call to Migration Manager Server object to convey binary content of the Migrant along with its serialized backup object. Succeeding that, Migration Manager Server in turn will accommodate and restart the migrated OSGi component in its host. Clearly, all services inside the OSGi bundle will get back its captured session data and continue theirs temporarily suspended executions. Finally, Migration Manager Client is responsible to remove the original copy of the migrated OSGi component and the connection between two hosts is disconnected.

### 6.6 Solution for migration concept

My approach to OSGi components migration in the OSGi Service Platform has to consider a number of fundamental design decisions for migration concept, which are discussed below.

#### 6.6.1 Migration Decision

The main idea here is to build a framework where OSGi services or objects residing inside an OSGi component can be triggered externally to migrate by Bundles Manager. This goal can be reached by modifying a specific entry of the OSGi bundle's properties and using tracking mechanism provided by the OSGi's `ServiceListener` model, or the specification of `ServiceTracker` in other words. More in detail, the unique `Activator` class of each migrative OSGi bundle must extend the base class, `AbstractMobilityActivator`, providing a function so called `suspendServices()`. The `suspendServices()` function takes its responsibility to change the bundle properties when it is called. And the modification subsequently informs
Migration Manager that the bundle is ready to be migrated.

Migration Manager consists of Migrative Service Tracker service that is an instance object of the Service Tracker class. The OSGi specification defines an utility class, Service Tracker, that makes a tracking process over registration, modification, and unregistration of OSGi services. Moreover, the Service Tracker class is customizable, therefore it is possible to extend its default implementation to meets typical needs defined by a Service Reference object and by a filter expression based on LDAP Search Filters. In the project, the instance of Service Tracker keeps track running states of migrative OSGi bundles by checking up on changing of their properties registered with the OSGi Service Registry. In order to allow the states tracking process provided by Migration Manager performs properly, the Activator class of every migrative bundle must also override its super class method suspendServices() and this function is also supplemented with some lines of code to suspend an execution flow of each of its internal service. The task can be completed in a straightforward way just by invoking the suspend() method of each service. The suspendServices() method performs a mission to change the OSGi component's properties.

A purpose to implement the two abstract functions, registerServices() and startServices(), is to register normal Java objects as OSGi services and start execution of those service respectively. These implementation tasks are optional and these are up to application programmers decisions.

Implementation details of the AbstractMobilityActivator class be found in the code snippet 1 in Appendix A.
6.6.2 State Saving

After interrupting execution of the migrative OSGi component, the next crucial step is to save session states of all OSGi services existing within the OSGi component. Many modern systems use the object serialization mechanism for this purpose. Serialization is a process of saving and conserving a runtime object's states to a fixed storage, like in a file. Those states consist of values of its all class member objects and variables. However, all information regarding method call stack and a program counter is missing. In the paper [32], authors declare firmly that the states transfer time can significantly be reduced by using Java externalization rather than serialization. Regarding to object serialization mechanism, a runtime object is to be marked as Serializable if its all class member variables and objects are also wrapped and marked as Serializable. Sometimes it is inefficient to serialize all class member variables because values of some variables are not required to be saved. Conversely, externalization allows the application programmer to write their own object transfer policies by saving only information necessary for rebuilding object graphs. “Externalization may be until 40% faster than serialization” [32]. In addition, by using serialization, most of the language level information required to re-establish the program states can not be captured. Especially, re-establishing states of an individual thread residing in multi-threads program is not trivial if we use serialization mechanism because we need to have an interface of serialized object's class in order to transform or type cast a binary or XML representation of this object stored in some flat file back to an instance of this object. In most case, we do not have this interface in a destination host before a mobile program is already transferred.

There is also another approach, Java Reflection, can be used to save values of class member variables at runtime. And this approach is claimed as a solution for the future work of the paper [22] that provides strong migration implementation for multi-threads agents system based on source-code preprocessor. They want to get rid of some limitations caused by source-code preprocessor and using reflection classes Java offers to do some parts of transformation at runtime is a feasible option [22]. Learning from advantages provided by such good ideas, concept of saving session states of each OSGi service, or a special Java object, in this paper can be realized in the combination form of Externalization and Reflection techniques. Acquiring knowledge from the way how to save execution states of a mobile program provided by the source-code preprocessor approach, it is absolutely possible to reuse the concept of backup object that stores all necessary relevant states information of the mobile program in the system. The backup object in a type of Java Dictionary can be considered as the most appropriate one because an object in Dictionary type can provide a way to store every kind of object/variable in a simple form of key/value. Especially, one object that is an instance of the Dictionary class can be dwelt inside another Dictionary object. Supposing that we have a collection of backup objects in Dictionary type of every OSGi service, we can put them all in one wrapper or cover-all backup object in a form of Dictionary type too. Intuitively, respecting to an OSGi component having many services within, it is completely easy to collect session sates of each of its individual OSGi service and pack them in a backup object and finally set them inside a cover-all backup object having Dictionary type for the future re-extracting and re-establishing session states purpose. In the paper work, it is only possible to capture and re-instantiate session states of a specific OSGi service if its main class extends the base
class MobilityAbstractClass. Implementation details of MobilityAbstractClass can be found in the code snippet 2 in Appendix A.

The MobilityAbstractClass provides two substantial functions that can be overridden within its sub class, getSessionData() and reinstantiateSessionData(). Java Reflection techniques is utilized to implement these two functions in order to capture and re-establish current values of all class member variables in Java Primitive Types when an OSGi component is suspended. Note that those class member variables must be set and get their values by using getter/setter methods. In Java object oriented language, getter/setter is a commonplace feature and application programmers should follow the pattern in order to make his code to be more comprehensive, readable and professional. In fact, it is quite straightforward to extend these two functions to support not only Java Primitive Types but Complex Data Types also.

Another benefit of obeying getter/setter pattern is that programmers can explicitly point out explicitly which class member variables should be captured and re-instantiated. A possibility to support capturing and re-establishing values of class member variables of Java Complex Data Types can be deferred to application programmers choice. Certainly, an application programmer can also override easily operations of capturing and re-instantiating values of class member variables provided by these two functions. In a case when some class member variables coming without getter/setter methods, current values of those variables can not be captured and re-instantiated accurately. However, application programmer can easily perform an action of supplementing for these missing stuffs by adding some lines of code in a really straightforward manner thanks to the nice feature, storing and to extracting object values in key/value form, of Java Dictionary class. Note that overriding these two functions of the MobilityAbstractClass class in such manner by invoking super.getSessionData() or super.reinstantiateSessionData() is not compulsory. However, application programmer is highly recommended to override these two functions in the appropriate way to save his contributed efforts by providing each necessary class member variables with getter/setter methods. A sample code of a Java class whose instance object's session states that can be captured and re-established at runtime is demonstrated in the code snippet 3 in Appendix A.

In addition, an OSGi component contains of many OSGi runtime services, therefore it is compulsory to capture and somehow re-establish session states of its all services independently. And the task can be done in an uncomplicated manner because it is possible to identify each OSGi service based on its unique Service ID. Imagining that we can place a backup object representing all unavoidable relevant states information of the OSGi service in a cover-all backup object of the whole OSGi component with its Service ID as a unique key. In a contrary manner, on the basis of unique Service ID of each OSGi service within the specific OSGi component, it is also really simple to extract their separated backup object from the cover-all backup object.

### 6.6.3 Transferring of Code and State Information

Once a destination host is identified successfully and an OSGi component and its all internal services are suspended as well as those OSGi services session states are captured and packed in a
cover-all backup object, the OSGi component's code fragments and the cover-all backup object are serialized to streams of bytes and transferred to the destination host through an established TCP connection at the transportation layer. Serialization process can be simply done because OSGi component code is stored in a particular Jar file and converting binary content of the Jar file to a stream of bytes is straightforward. The backup object has its type of Dictionary and we all know that Dictionary object is **Serializable**.

An instance object of the File Packet class represents for a given Jar file containing binary content of one identified OSGi bundle. This class provides two main functions `readIn()` and `getData()` allowing to parcel binary code of the OSGi component into a **Serializable** object and it can be easily moved over network to a target host. Code snippets of File Packet class readers can find in the code snippet 4 in Appendix A.

The architecture system introduced in this paper contains the Migration Manager component that is available on every host. Migration Manager performs two main tasks: answering requests for migrating and working as receiver for a migration of data stream. It is also responsible for the continuation of the migrated OSGi component by installing, restarting, and re-instantiating saved session states of the migrant in the destination host.

As it is already mentioned in the section 3.6 of the paper that a dependencies graph between OSGi bundles allocated in a same host is really huge, therefore Migration Manager also takes a charge of checking if all dependent OSGi packages or bundles of the migrated OSGi component are already inhabited in the destination host or not. Migration Manager will transfer all missing dependent packages to the destination host and install them in advance. The task must be compulsorily completed, otherwise the migrated OSGi component can not be installed and restarted to run correctly in the destination host. The mechanism of finding which OSGi packages and bundles providing shared objects or classes used by the migrated OSGi component and checking their existence in the target host will be depicted in detail in the next section 6.2.4. After successfully transmitting the suspended OSGi component, Migration Manager will destroy the original bundle just being migrated in the source host.

### 6.6.4 Handling of OSGi bundles dependencies

OSGi specification comes up with a concept of dependencies between OSGi bundles. Bundles can export packages to other bundles. This exporting creates a dependency between a bundle exporting a package and a bundle using the package. If an OSGi bundle is moved from one place to another place, all other bundles exporting packages or services that are used by the migrant must be also transferred. However, it is not trivial to find out all of these dependent bundles for a specific bundle. Fortunately, the previously mentioned Package Admin service in the section 5.1.4 can provide an access to the internal OSGi framework package sharing mechanisms. By utilizing Package Admin service, it is able to find all required bundles of a specified bundle that need to be migrated. The algorithm realizing this idea is sketched out in the code snippet 4 in Appendix A.

Package Admin class provides the `getExportedPackages()` method that returns a list of Exported Package objects for each package that a given bundle exports. Moreover, it is also capable to create a list of all importing bundles of a given bundle by invoking
getImportingBundles() method of the Bundle class. Therefore, by going through the returned list of those importing bundles, it is possible to determine whether the specified bundle imports services or packages granted by which element of the list by making a comparison operation based on a unique BundleID of each bundle. This job helps to collect a list of all directly dependent bundles of the specified bundle. It becomes now trivial to set a boundary around all bundles which the specified migrative OSGi bundle depends on. In the architecture system proposed in the paper, those bundles will be transferred in advance before the migrative OSGi component will be relocated. It insists on a declaration that the migrative OSGi bundle can be restarted in the destination host properly.

6.6.5 Continuation of Executions and State Re-establishing

The fundamental idea is that every service of a transferred active OSGi component must continues its execution with its states information placed in a cover-all backup object. In my own approach, it is simple to take out an object that represents session states of each individual service from the cover-all backup object. The reason is that a backup object of a specific service can be placed in the cover-all backup object of the shelter OSGi component by using its Service ID. Conversely, it is able to extract any object out from the cover-all backup object by passing an appropriate key value, or a value of Service ID in other words. With a declaration of the reinstantiateSessionData() method of the super class MobilityAbstractClass, it could be facile to insert relevant states information back to the service and the service can go on with correctly saved session states in its execution.
Chapter 7  Design and Implementation

Detailed designs and implementations of the system's main components will be sketched out as below in the chapter.

7.1 User Migrative OSGi Component

Migrative OSGi component can also be called as a migrant in the paper. Application programmers involve in an implementation phase of migrative OSGi component. First of all, it requires them to have a quite good background knowledge in OSGi area. At least, they have to manage themselves to write runnable OSGi bundles by following a couples of fundamental rules, namely explicitly declaring Activator class for a specific bundle, writing a correct manifest file under the pre-defined syntax, etc. Moreover, the implemented OSGi component must also comply with some other rules that are predefined specifically just for the system. For instance, the Activator class of the migrative OSGi bundle must extend the abstract class AbstractMobilityActivator. The main functionalities and responsibilities of the AbstractMobilityActivator are characterized in the section 6.6.1.

```java
public class Activator extends AbstractMobilityActivator implements BundleActivator {
    private MigratableClass1 migratableObject1;
    private MigratableClass2 migratableObject2;
    public void registerServices() {
        migratableObject1 = new MigratableClass1();
        props.put("service.pid", "object1");
        this.bundleContext.registerService(migratableClazzes1,
            migratableObject1, props1);
        migratableObject2 = new MigratableClass2();
        props.put("service.pid", "object2");
        this.bundleContext.registerService(migratableClazzes2,
            migratableObject2, props2);
    }
    public void suspendServices() {
        migratableObject1.suspend();
        migratableObject2.suspend();
        super.suspendServices();
    }
    /**
    * See missing parts of this class in source code of the project
    **/}
```

Within an implementation of the Activator class, application programmers should pay
attention to identify each Java object with its unambiguous Service ID while registering it as an ordinary OSGi service. The unique Service ID of the service allows the system to put and get a respective backup object representing its session states properly from the cover-all backup object without any difficulty and operations of putting and getting a backup object of each service is completely transparent to application programmers and end users at runtime. In order to enable Bundles Manager component, that will be taken into account in the next section 6.3.5, to explicitly suspend an execution flow of an OSGi bundle, the suspendServices() method must be overridden in a proper way inside the Activator class. Obviously, application programmers must also follow some other aforementioned regulations while coding Java classes of the OSGi services. For example, the classes must extend the abstract class MobilityAbstractClass. The requirements are already stated in the section 6.6.2.

7.2 Migration Manager Component

In this part, readers can have chance to take a closer gaze at the internal structure of the Migration Manager component. In addition, technical specifications of its elements will be characterized in many detailed aspects.

7.2.1 Constituents of Migration Manager

Figure 8: Constituents of Migration Manager Component

Migration Manager Server service of a specific Migration Manager component and Migration Manager Client service of another Migration Manager component establish a
communication pipe between the server and client sides. And the established connection is used to transmit a specific OSGi component's binary code and its relevant states information (See details in the sections 7.2.2 and 7.2.3).

With Http Stub Downloader, RMI Dynamic Class Loading can be employed in my project. RMI Dynamic Class Loading is a nice feature of RMI that allows implemented binary code of the server object can be dynamically downloaded from the server side to the client side through the HTTP protocol (See details in the section 7.2.4).

Migrative Service Tracker Customizer extends the OSGi built-in Service Tracker Customizer. An instantiated object of this class provides a monitoring mechanism to keep tracking if some OSGi bundle's state is changed to “Ready To Be Migrated”.

File Packet represents a file object that can be sent and recreated on another system. Especially, File Packet acts as a representative for a given Jar file having a binary content of some indicated OSGi bundle.

Bundle Properties is responsible for loading default system properties of Migration Manager Server and Migration Manager Client services.

An implementation of Service Bean envelopes all substantial informations of the Migration Manager Server's remote reference. An instance object of the Service Bean implementation is useful when a remote reference of Migration Manager Server needs to be moved over network and installed in a remote host.

7.2.2 Migration Manager Server

**Figure 9: Migration Manager Server class diagram**

Migration Manager Server is a regular OSGi service. It plays the central role in the whole system. Migration Manager Server and Migration Manager Client communicate with each other via
RMI technology. From a view to RMI-based system, Migration Manager Server is the server side object and Migration Manager Client acts as a client side object. When a connection between these two object is set up, Migration Manager Server adds a remote reference of the Migration Manager Client to its list of clients by invoking a method `addRemoteClientToServer()`.  

```java
public void addRemoteClientToServer(ManagerClientInterface managerClient)
{
    if( !clientsList.contains( managerClient ) )
    {
        clientsList.add( managerClient );
    }
    /**
     * See missing parts of this method in source code of the project
     **/
}
```

Due to the fact that the connection between Migration Manager Server and Migration Manager Client is occasionally cut off, it sounds promising if we can deal with challenging issues of disconnected connection. In my approach, by taking advantages of RMI Callback feature, a remote reference of Migration Manager Server object is transferred and registered in every client that establishes a communication channel with Migration Manager Server.

```java
public void addRemoteClientToServer(ManagerClientInterface managerClient)
{
    ServiceReference[] srefs = rmiServiceTracker.getServiceReferences();
    for( int i = 0; i < srefs.length; i++ )
    {
        registerServiceOnRemoteOSGi( srefs[i], new ManagerClientInterface[] {managerClient} );
    }
    /**
     * See missing parts of this method in source code of the project
     **/
}
```

The method `registerServiceOnRemoteOSGi()` is appealed whereas Migration Manager Server want to register its remote reference as a normal OSGi service in all hosts where its Migration Manager Clients are allocated. All unavoidable data of the remote reference is compressed within an instance object of the Service Bean class.
private Object registerServiceOnRemoteOSGi( ServiceReference reference,  
Object [] rmiClientsList ){
      Remote migrationManagerServerService =  
      (Remote)bundleContext.getService( reference );

      String serviceClassName = rmIService.getClass().getName();  
      ServiceBean bean = new ServiceBeanImpl();  
      bean.setHashCode( reference.hashCode() );  
      bean.setClasses( clazzes );  
      bean.setRMIService( migrationManagerServerService );  
      bean.setProperties( properties );  
      for( int j = 0; j < rmiClientsList.length; j++ )  
      {  
          ((ManagerClientInterface)
              rmiClientsList[j]).registerRemoteService( bean );  
      }

      /**  
      * see missing parts of this method in source code of the project  
      **/  
}

Note that one Migration Manager Server object can maintain many connections  
simultaneously with multiple instances of Migration Manager Client. It is really practically useful  
when there is a need to increase the concurrency capacity of the system. Especially, these  
concurrent connections do not block or impact actions of each other because RMI supports multi-  
threading inside its internal mechanism. When Migration Manager Client terminates a connection,  
the methods removeRemoteClientFromServer() is called in order to delete its remote  
reference from the list of connected clients in the server side. Consequently, Migration Manager  
Server will also remove its remote reference from the client side by making a request to a method  
unregisterServiceOnRemoteOSGi(). These two method calls  
RemoveRemoteClientFromServer() and unregisterServerOnRemoteOSGi() are  
executed in an reserve manner with the addRemoteClientToServer() method and the  
registerServiceOnRemoteOSGi() method respectively.

Four private methods, namely startRMIRegistry(), setup(), open() and  
close(), are used to set up the Migration Manager Server service to act as a normal RMI object  
in server side. Normally, from points of view of Java programmers, they run RMI applications by  
explicitly starting RMI Registry and RMI applications from a console. Unfortunately, it is  
impossible to run any RMI application in form of an OSGi bundle from a console. In the paper, the  
RMI Registry is located and RMI object is bound to the RMI Registry by invoking explicitly two  
methods startRMIRegistry() and setup() of the Migration Manager Server object.
private Registry startRMIRegistry(){
    rmiRegistry = LocateRegistry.createRegistry(    
            this.RMIRegistryPort );
    /**
     *  See remaining parts of this method in source code of the project
    **/
}

public boolean setup( int RMIRegistryPort )
{
    rmiRegistry = LocateRegistry.getRegistry(    
            this.RMIRegistryPort );
    rmiRegistry.bind( "MigrationManagerServer", this );
    /**
     *  See remaining parts of this method in source code of the project
    **/
}

These two methods hides all RMI related essence from application programmers views. RMI mechanism in the underlying communication layer therefore is completely transparent to application programmers. They do not be aware of whatever communication technology is currently utilized.

The isBundleExist() method is invoked upon a request when any Migration Manager Client wants to make a query to discover if some specified OSGi bundle is already resided in the OSGi framework in the server side.

A binary content of any OSGi bundle and its session states will be transmitted from a client side to a server side. Subsequently, the OSGi component is reinstalled in the OSGi platform where the Migration Manager Server service is settled. BundleContext can help us to install a new bundle from an InputStream object via its installBundle() method. It is necessary to give the BundleContext object an actually binary content of the OSGi component extracted from a transferred FilePacket object in a form of a byte stream. Re-instantiating all essential session states for each service inside the OSGi component is the next step. All of these tasks are carried out completely after the method receiveAndInstallFile() of Migration Manager Server class is appealed.

public void receiveAndInstallFile( FilePacket packet, Dictionary data) throws Exception
{
    ByteArrayInputStream inputStream = new
            ByteArrayInputStream(packet.getData());
    Bundle bundle=
            this.bundleContext.installBundle(packet.getName(), inputStream)
    ServiceReference[] registeredServices =
            bundle.getRegisteredServices();
}
for(int i=0; i<registeredServices.length; i++)
{
    Object obj =
        this.bundleContext.getService(registeredServices[i]);
    if(obj instanceof MobilityAbstractClass){
        MobilityAbstractClass migratedObject =
            (MobilityAbstractClass) obj;
        Dictionary dataForEachObject =
            (Dictionary)data.get(registeredServices[i].getProperty
                ("service.pid"));
        migratedObject.ReinstantiateSessionData(dataForEachObject)
    }
}
/**
* See remaining parts of this method in source code of the project
**/
Migration Manager Client to start up RMI Registry, to bind itself with the created RMI Registry and finally to create a communication link with the specified Migration Manager Server object in the server side. The following methods, namely close(), disconnect() and freeze(), perform opposite functions to the open(), connect() and startOpening() methods respectively.

Whenever Migration Manager Client object send a connecting request to Migration Manager Server, the object in the server side will then automatically transfer its remote reference to the client side by making an invocation call to the method registerRemoteService() of Migration Manager Client object. Consequently, the remote reference will be registered as a standard OSGi service and the OSGi service plays a role as a smart proxy in the client host. A few on going projects also employ this technique due to its many advantages. The main advantage is that by using the smart proxy of Migration Manager Server service, Migration Manager Client service can treat Migration Manager Server service in the server side with the same manner as all regular local OSGi services. Right after Migration Manager Server service disappears in the remote platform or it is disconnected from a network, the Migration Manager Client service will be signalled about the event. After that, the Migration Manager Client object can reconnect to the corresponding Migration Manager Server in an automatic manner. Certainly, an ability of Migration Manager Client to rebind automatically with Migration Manager Server is useful and applicable especially with a frequently disconnected network like WLAN.

Before transmitting code and session states of an OSGi component to a target host, it is compulsory to define explicitly an IP address of the destination and to pass the IP value to Migration Manager Client object, so that Migration Manager Client can create a proper connection link to Migration Manager Server. In the system, Bundles Manager component receives this duty. Obviously, there are two possibilities. Regarding to the first possible solution, Bundles Manager bundle may hold a service reference of Migration Manager Client service and it can invoke directly a specific method of Migration Manager Client object to pass the predefined IP as a parameter. However, it requires someone who implements Bundles Manager to understand the implementation of Migration Manager Client and to have a knowledge how to get a service reference of Migration Manager Client service. Clearly, the first solution is not flexible. Configuration Admin service may provide an alternative and more flexible solution that allows an apparent separation between two modules, Migration Manager and Bundles Manager, with regards to implementation aspects. Programmer coding the Bundles Manager component should not be aware of an insight structure of Migration Manager but he have only to be acquainted with a basic feature provided by Configuration Admin. Configuration Admin service allows configuration data of deployed bundles to be defined and those bundles will receive that data when they are active in the OSGi Service Platform. The IP address of the destination host can be regarded as configuration data of Migration Manager Client object. Updated() method is called when Configuration Admin sends Migration Manager Client object its configuration data and apparently the configuration data can be changed dynamically.

After the connection is based prosperously and Migrative Service Tracker listening seamlessly on the Service Registry acquires an event signifying about the “ready to be migrated” state of some OSGi component, Migrative Service Tracker will make an invocation to
the method `migrate(ServiceReference)` of Migration Manager Client. This called method allows a bundle identified by its `ServiceReference` transferred together with its saved states and all of its relevant bundles. Detailed implementation of this method is demonstrated in the code snippet 5 in Appendix A.

### 7.2.4 HTTP Stub Downloader

![Diagram of Http Stub Downloader class diagram]

RMI requires that client uses a stub class to connect to a remote VM. The stub class hides the details of network communication, sending method arguments and waiting for a return value. Clients need a way to guarantee that stubs are available. Installing the stubs on the clients side is not always viable because stubs are an implementation detail that might change over time. Fortunately, the RMI runtime system has a dynamic class loading facility that loads the classes which it needs while executing remote method calls. Dynamic class loading allows the clients virtual machines to find appropriate stubs at runtime without any special coding in the client. In order to adopt advantages of dynamic class loading mechanism to the system, Http Stub Downloader is developed. An implementation of the abstract class `Service Tracker Customizer` allows the system to keep track the Http Service. In OSGi Service Platform, Http Service can be treated as a light-weight HTTP Web Server. Http Stub Downloader class also has in its hand an instance object of `HttpContext` class that is really important for the proper operation of Http Service. By the way, the server side code base can be installed properly and clients can download stub classes based on the indicated code base.

Migration Manager Server's detail can be changed without any notification to Migration Manager Client. However, the client object can get the most recently generated stub class of the Migration Manager Server object at the time when a connection to the server is established.
Apparently it is truly useful because Migration Manager Server's detail can be altered to make any coding enhancement without making any influence to Migration Manager Client's detail. Conversely, any modification with implementation of Migration Manager Client class does not impact operation of Migration Manager Server. Incidentally, the system is highly extendible and scalable by using the dynamic class loading mechanism.

7.3 Bundles Manager Component

The default Bundles Manager's GUI and its basic functionalities are described in words in this chapter. This chapter is really worth to read for end users who want to use any distributed system elaborated on top of the system. Application programmers may also get many benefits from detailed descriptions of the Bundles Manager. This means that they can enhance the Bundles Manager's outlook as well as its functionalities.

7.3.1 Bundles Manager Functionalities

![Figure 12: Graphical User Interface of Bundles Manager](image)

With GUI (Graphical User Interface) provided by the Bundles Manager component, any potential end user can easily interact with the developed system. He can establish a connection channel with one target host by identifying its IP address. After that, he can select a particular OSGi bundle from a list of active bundles in the OSGi service platform. Finally, he can suspend the component execution and migrate it to the determined destination host. Due to the fact that bundles can appear (or install), disappear (or reinstall) and change their states (active, resolved, etc.), Bundles Manager offers end users with a functionality to obtain the list of most recently active bundles.

Of course, it is unnecessary for end users to have any knowledge of all underlying layers operating mechanisms. In other words, implementation of migration concept in the system is completely transparent to end users. With regards to application programmers, it is slightly straightforward for them to alter the implementation of Bundles Manager even they just have really a limited knowledge about the underneath communication layer, especially about detail implementation of Migration Manager component. Inside the system, relationships between components in different layers is really loose. To state the matter differently, any change in
implementations of components at the underlying communication layer does not influence towards implementations of components at the above application layers, namely Bundles Manager, Migrative OSGi Bundles and conversely.

7.3.2 Class Diagram

![Figure 13: Bundles Manager class diagram](image)

Bundles Manager class extends the JFrame class. It contains instance objects of the JButton class, namely SuspendButton, RefreshButton and LocateButton. Each class object is bound to an instance object of a class extending the base class ActionListener such as SuspendButtonHandler, RefreshButtonHandler and LocateButtonHandler respectively. These three classes are constructed as inner classes of the Bundles Manager class, therefore they can access public and private class member variables and objects of Bundles Manager class. Bundles Manager initiate its all class member variables and objects inside its constructor.

7.4 Testing Results

To demonstrate how the system works, a simple OSGi component that provides a service calculating set of Fibonacci numbers is implemented. In addition, a simple packages that offers shared objects that are used by the migrative OSGi component is also coded. We can take a look at the next screen shots to get an overview of the system's operations.
First step: Before the migrative OSGi component is migrated.

Figure 14: OSGi Platform at client side before migration

In the OSGi Service Platform at the client side before the migration routine is performed, there are some components: Migration Manager (Manager_Client), Bundles Manager (GUIManager), Migrative OSGi Component (Migratable_Object) and its dependent package (DependencePackage). The migrative OSGi bundle has a computation service that calculates the Fibonacci numbers (see figure 14).

Second step: Locate the destination host

Figure 15: Bundles Manager GUI

Figure 15 illustrates out the Graphical User Interface of the Bundles Manager component. We can locate the destination host (IP 141.76.178.246) where the migrative OSGi component will be accommodated after the migration routine is performed. After that, we can select the migrative
OSGi component from the list of active bundles. Subsequently, we can suspend executions of services inside the selected bundle and migrate the bundle to the destination host by clicking on the button named Suspend and Migrate.

**Third Step: Client side after the migration routine is finished**

![Figure 16: OSGi Platform at client side after migration](image)

From figure 16, we can see that the selected bundle has been removed and it has been transferred to the destination host together with its dependent package that is not removed. The computation service inside the migrated bundle has stopped its calculation routine at the Fibonacci number 12\textsuperscript{th} with a value 144.

**Final step: Server side after the migration routine is finished**
From the figure 17, we can easily realize that the migrated OSGi component has been reinstalled together with its dependent package. After that, the computing service inside the migrated bundle restarts its calculation routine from the Fibonacci number 12\textsuperscript{th} and it continues generating the next Fibonacci numbers. This means the computing service can be executed continuously from the point where it is suspended at the client side.
Chapter 8 Conclusion and Outlook

8.1 Summary

In the paper, in order to find out the most appropriate solution that is feasible to realize migration concept of OSGi component, many existing approaches in the research area of mobile code have been investigated. Many challenges posed by each of the approaches have been examined and analysed in terms of its capability to migrate OSGi component with its saved states. After evaluating complexity and feasibility of each approach in concerning with constraints in implementation time and contributed efforts, the system which supports stateful weak migration concept of OSGi component has been developed. The system represents an operational runtime environment that is both conceptually and physically lightweight. And it enables application programmers to build any distributed application that is developed based on the mobile code concept and on top of OSGi platform just with little efforts. Migrative OSGi components as parts of the distributed application can be transferred across a network together with its captured session states. Right after it reaches any target host, the system will give it back its saved states, therefore it can restart its execution with exactly the same session data when it was suspended previously. From an external view, execution of the migrated OSGi component seem to be partly continuous.

8.2 Future Works

There are two main future tendencies that are really worth to be studied and implemented. First, the final small demonstration is coded during the project's implementation phase to prove that the system can work out and to substantiate with evidence that the saving/restoring states concept has been partially actualized. However, regarding to real world needs, this demonstration is meaningless because it can not be put to use in any application. Hence it is much more meaningful if any business-oriented real-world application can be designed and developed on top of this system. Second, the saving/restoring states concept has been not wholly accomplished because only values of variables and objects in a class scope can be captured and re-instantiated. Therefore, deeper and more intensive research should be carried out to verify whether a strong migration concept can be finally realized or not. From my point of view, there are two possible candidates that can be taken into account: byte-code instrumentation with a support of Dynamic AOP concept and Continuation concept and Javaflow until its matures in the near future.

8.3 Personal Evaluation

At the end of thesis, I would like to give a personal evaluation about my work. Within the time schedule of this thesis (approximately 200 working hours in maximal 6 continuous months), it took my about 2 months to learn basic concepts and technologies. During these two months, many literatures and paper works have been read and consequently a wide range of knowledge in the mobile code area is accumulated.
With many worthy helps and advices of my supervisors, during the design phase, I did not meet any unsolvable problem.

During the implementation phase, I have met some troubles while implementing software components of the system via using APIs of some OSGi standard services like Package Admin, Configuration Admin, HTTP Service correctly and sufficiently. Fortunately with helps of many experts in a network of OSGi development, the problems are completely solved out.

Regarding to gained achievements, I would consider this thesis to be successful for reaching many objectives (section 1.2). Although source code of the project has not been well-documented but still comprehensive for readers after they take a quick glance at the overall architecture of the system.

Concerning to the used technologies, I have learned and studied many useful stuffs related to OSGi standard services like HTTP Service, Package Admin, Configuration Admin. In addition, I have acquired much deeper knowledge in area of developing an application based on the OSGi Service Framework. I have to admit that it is not trivial and not easy to develop the system on a basis of the OSGi Framework unless having really good programming skills and ability to learn quickly new techniques. RMI is used for a development purpose of communication channel. RMI is not new but it is the most appropriate communication protocol under the specific scope of the project.

Even performance of the system has not been verified and tested systematically by making experiments and any comparison with other implementation, it is likely believable that the system can work really well thanks to the well-known RMI's good performance capability.
Literatures


Appendix A: Code snippets

```java
public abstract class AbstractMobilityActivator {
    private ServiceRegistration registry;

    public void suspendServices(){
        Hashtable states = new Hashtable();
        states.put("rmi.migratable",
            AbstractMobilityActivator.class.getName());
        if(this.registry != null)
            this.registry.setProperties(states);
    }

    public abstract void startServices();
    public abstract void registerServices();
    public void setRegistrationReference(ServiceRegistration reg){
        this.registry = reg;
    }
}
```

**Code snippet 1: AbstractMobilityActivator**

```java
public abstract class MobilityAbstractClass{
    public abstract void suspend();

    public Dictionary getSessionData() throws Exception {
        Hashtable hashtable = new Hashtable();
        /**
        * The remaining parts of the method can found in source code
        * of the project
        **/
        return hashtable;
    }
}
```

public void ReinstantiateSessionData(Dictionary sessionData) throws Exception {

/**
  * Content of this method is so long to be showed in the report.
  * Readers can take a look at source code of the project
  **/}
}

Code Snippet 2: MobilityAbstractClass

```java
public class MigratableClass extends MobilityAbstractClass implements Runnable{
    /**** session data of migratble class service***********/
    private int counting = 0;
    private BigInteger fibNumber = new BigInteger(String.valueOf(0));
    /******************************************************/

    public void run() {
        try{
            while(!isStopped){
                try{
                    Thread.sleep(1000L);
                }catch(Exception e){}
                counting++;
                System.out.println("Now: " + counting);
                fibNumber = fibNumber.add(new BigInteger(String.valueOf(1)));
                System.out.println("The Fibonacci number " + counting + "th" +
                "has a value: " + fib(fibNumber).toString());
            }
        }catch(Exception e){}
    }
}
```
public Dictionary getSessionData() throws Exception{
    Hashtable hash = new Hashtable();
    hash = (Hashtable)super.getSessionData();
    hash.put("fib", fibNumber);
    return hash;
}

public void ReinstantiateSessionData(Dictionary sessionData) throws Exception{
    super.ReinstantiateSessionData(sessionData);
    this.fibNumber = (BigInteger)sessionData.get("fib");
}

public int getCounting() {
    return counting;
}

public void setCounting(int counting) {
    this.counting = counting;
}

/**
 * Missing parts of this class can be found in source code of the
 * project
 **/
Bundle[] importingBundles =
    exportedPackages[p].getImportingBundles();
if (importingBundles == null) continue;
for(int l = 0; l < importingBundles.length; l++){
    if(importingBundles[l].getBundleId() == bundle.getBundleId())
    {
        listResult.add(bundles[i]);
    }
}
/**
 * Try catch block and Logging routines can be found in source
 * code of the project
 **/Bundle[] bundlesList = new Bundle[listResult.size()];
System.arraycopy(listResult.toArray(), 0, bundlesList, 0,
    listResult.size());
return bundlesList;

public void migrateBundle(ServiceReference reference) throws Exception{
    /**
     * Make a loop throughout a list of the migrated component's
     * packages or bundles. Call appropriate method to transfer them all
     * the specified destination host.
     * See the missing parts of the method in source code of the
     * project
     **/
    /**
     * Extract states of each service inside the bundle and save them
     * into Dictionary object
     **/
    String location = reference.getBundle().getLocation();
    Bundle migratedComponent = reference.getBundle();
    Hashtable sessionData = new Hashtable();
    ServiceReference[] registeredServiceReferences =
        migratedComponent.getRegisteredServices();
    for(int i=0; i<registeredServiceReferences.length; i++){
        Object migratedObj =
            bundleContext.getService(registeredServiceReferences[i]);
        MobilityAbstractClass migratedObject = null;
        if(migratedObj instanceof MobilityAbstractClass){
            migratedObject = (MobilityAbstractClass) migratedObj;
Dictionary sessionDataForEachObject =
    migratedObject.getSessionData();
    sessionData.put( registeredServiceReferences[i].getProperty
    ("service.pid"), sessionDataForEachObject);
}

/**
 * Transfer the migratable bundle
 **/
    filein = new FilePacket(location.trim().substring(5));
    filein.readIn();
    boolean isSucceed = false;
    int retries = 0;
    while((retries < RemoteConstants.NUMBER_OF_RETRIES)
        && (isSucceed == false))
    {
        isSucceed = migrationManagerServer.receiveAndInstallFile
            (filein, sessionData);
        if (isSucceed == false){
            retries++;
            Thread.sleep(500L);
            continue;
        }
    }
    /**
     * Remove the original bundle after the transmission succeeded.
     **/
    filein = null;
    bundleContext.ungetService(reference);
    bundle.uninstall();
}

Code Snippet 5: Method migrateBundle() of Migration Manager Client
Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

Date and Signature
Vu Duc Lam